НАЦІОНАЛЬНИЙ ТЕХНІЧНИЙ УНІВЕРСИТЕТ УКРАЇНИ
«КИЇВСЬКИЙ ПОЛІТЕХНІЧНИЙ ІНСТИТУТ імені ІГОРЯ СІКОРСЬКОГО»

Теплоенергетичний факультет

Кафедра автоматизації проектування енергетичних процесів і систем

До захисту допущено
Завідувач кафедри
_____ О.В. Коваль
(підпис) (ініціали, прізвище)
“___”_____2020р.

ДИПЛОМНА РОБОТА

на здобуття ступеня бакалавра

з напряму підготовки 121 Інженерія програмного забезпечення

на тему Бібліотека підтримки окремих баз даних різних підрозділів університету в мікросервісній архітектурі

Виконав: студент 4 курсу, групи ТВ-з61

Туляков Євгеній Вячеславович _____
(прізвище, ім’я, по батькові) (підпис)

Керівник доцент Смаковський Денис Сергійович _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Консультант _____ _____ _____
(назва розділу) (вчені ступінь та звання, прізвище, ініціали) (підпис)

Рецензент_____ _____
(посада, вчене звання, науковий ступінь, прізвище та ініціали) (підпис)

Засвідчую, що у цій дипломній роботі немає запозичень з праць інших авторів без відповідних посилань.
Студент _____
(підпис)

Київ – 2020 року

Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”

Факультет теплоенергетичний

Кафедра автоматизації проектування енергетичних процесів і систем

Рівень вищої освіти перший рівень

Напрям підготовки: 121 Інженерія програмного забезпечення

Спеціалізація: Програмне забезпечення розподілених систем

ЗАТВЕРДЖУЮ
Завідувач кафедри
_____ О.В. Коваль
(підпис)
”___”_____2020р.

ЗАВДАННЯ
на дипломну роботу студенту

Тулякову Євгенію Вячеславовичу
(прізвище, ім’я, по батькові)

1. Тема роботи Бібліотека підтримки окремих баз даних різних підрозділів університету в мікросервісній архітектурі

керівник роботи доцент Смаковський Денис Сергійович
(прізвище, ім’я, по батькові науковий ступінь, вчене звання)

затверджена наказом вищого навчального закладу від ”__”___ 202__р. № __

2. Строк подання студентом роботи _____
3. Вихідні дані до роботи _____
_____
_____

4.Зміст розрахунково-пояснювальної записки (перелік питань, які потрібно розробити)_____
_____
_____
_____
_____
_____
_____
_____
_____

5. Перелік ілюстративного матеріалу

_____
_____
_____
_____
_____

6. Консультанти розділів роботи

Розділ	Прізвище, ініціали та посада Консультанта	Підпис, дата	
		завдання видав	завдання прийняв

7. Дата видачі завдання ”___”_____201___ р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів виконання дипломної роботи	Термін виконання етапів роботи	Примітки
1.	Затвердження теми роботи		
2.	Вивчення та аналіз задачі		
3.	Розробка архітектури та загальної структури системи		
4.	Розробка структур окремих підсистем		
5.	Програмна реалізація системи		
6.	Оформлення пояснювальної записки		
7.	Захист програмного продукту		
8.	Передзахист		
9.	Захист		

Студент _____ Туляков Є.В._
(підпис) (прізвище та ініціали,)

Керівник роботи _____ Смаковський Д.С._
(підпис) (прізвище та ініціали,)

ВІДГУК

керівника дипломної роботи

освітньо-кваліфікаційного рівня „бакалавр”

виконаної на тему : Бібліотека підтримки окремих баз даних різних підрозділів університету в мікросервісній архітектурі

студентом Туляковим Євгенієм Вячеславовичем

Тема є актуальною з точки зору активного росту кількості мультиклієнтних програм, а також необхідності розвитку універсальних веб додатків для різних факультетів університету.

Головна ціль дипломної роботи – аналіз існуючих рішень та розробка бібліотеки для підтримки окремих баз даних різних підрозділів університету в мікросервісній архітектурі. Бібліотека має бути універсальною та інтегруватися з сучасними технологіями. Розроблена в інтересах науково дослідницької роботи кафедри.

Дана дипломна робота повністю відповідає поставленому завданню. Тема розкрита у повній мірі.

Студент був досить самостійним під час виконання дипломної роботи. Студент на практиці знайомий з сучасними технологіями та рішеннями, тому вміло обрав необхідні інструменти для вирішення задачі дипломної роботи.

Робота у достатній мірі проілюстрована прикладами та поясненнями. Оформлення роботи відповідає вимогам до бакалаврських робіт.

Робота виконана на високому науково-технічному рівні та допускається до захисту з оцінкою 95 балів (“**відмінно**”). Туляков Є.В. заслуговує присвоєння рівня «бакалавр».

Керівник дипломної роботи

кандидат технічних наук, доцент	_____	Смаковський Д.С.
(посада, вчені звання, ступінь)	(підпис)	(ініціали, прізвище)

РЕЦЕНЗІЯ

на дипломну роботу

освітньо-кваліфікаційного рівня „бакалавр”

виконаної на тему : Бібліотека підтримки окремих баз даних різних підрозділів університету в мікросервісній архітектурі

студентом Туляковим Євгенієм Вячеславовичем

Підтримка мультиклієнтосні – досить комплексна та актуальна тема, про що свідчить активний інтерес громади розробників. Постійно вдосконалюються існуючі рішення та створюються нові. Що стосується створення універсальної бібліотеки, то у цьому існує реальна необхідність. Наразі функціонал підтримки мультиклієності дублюєтсья у кожному сервісі. Винесення цього функцоналу у окрему бібліотеку стало кроком до прискорення розробки програмного забезпечення та кроком до побудови сервісів за одним з принципів SOLID – єдиної відповідальності.

Тема роботи розкрита у повній мірі. У роботі висвітлено стан справ у теоретичному аспекті на сьогодні, а також специфіковано композиційний підхід для розв’язання поставлених задач. Побудовано та задокументовано архітектуру взаємодії мікросервісів системи університету, що виступає у якості демонстрації використання бібліотеки. Приведено достатньо прикладів та пояснень щодо роботи програми.

Випускна робота Тулякова Є.В. виконана на високому професійному та науково-технічному рівні, з дотриманнях всіх норм та вимог, що висуваються до бакалаврських робіт, має актуальне практичне застосування, тому заслуговує оцінки у 95 балів (“**відмінно**”).

Рецензент

кандидат технічних наук, доцент _____ Галкін Олександр Володимирович

(посада, вчені звання, ступінь) (підпис) (ініціали, прізвище)

АНОТАЦІЯ

до бакалаврської дипломної роботи Тулякова Євгенія Вячеславовича
на тему: «Бібліотека підтримки окремих баз даних різних підрозділів університету в мікросервісній архітектурі»

Дана дипломна робота присвячена розробці бібліотеки для підтримки мультиклієнтності у мікросервісах. В роботі зроблено аналіз видів мультиклієнтності, їх переваги та недоліки. Створено бібліотеку та систему для демонстрації її роботи. Приведені приклади роботи системи.

Серверна частина розроблялася з використанням мови програмування Java та технології Spring Boot, клієнтська – використовуючи такі технології: HTML, CSS, JavaScript, Thymeleaf; база даних – PostgreSQL.

Загальний об'єм роботи 92 сторінки, 33 рисунки, 27 таблиць, 3 додатки, 10 бібліографічних найменувань.

Ключові слова: база даних, таблиця, веб сервер, веб інтерфейс, мультиклієнтність, Java, факультет, бібліотека.

ABSTRACT

to the diploma thesis by Tuliakov Yevhenii Vyacheslavovich
on the topic «Library of support of separate databases of various divisions of university in microservice architecture»

This thesis is devoted to the development of a library to support multitenancy in microservices. The analysis of types of multitenancy, their advantages and disadvantages is made in the work. A library and a system for demonstrating its work have been created. Examples of the work are given.

The server part was developed using Java programming language and Spring Boot technology, the client - using the following technologies: HTML, CSS, JavaScript, Thymeleaf; database - PostgreSQL.

The total volume of work is 92 pages, 33 figures, 27 tables, 3 appendices, 10 bibliographic titles.

Keywords: database, table, web server, web interface, multitenancy, Java, faculty, library.

ЗМІСТ

ПЕРЕЛІК УМОВНИХ ПОЗНАЧЕНЬ

ВНЗ – вищий навчальний заклад

ЕОМ – електронна обчислювальна машина

ДСТУ – Державний стандарт України

БД – база даних

СУБД – система управління базами даних

Таблиці сутності – основні таблиці, в яких міститься інформація, що динамічно змінюється

Таблиці-словники – таблиці, в яких зберігається статична інформація, яка вноситься в БД один раз і може інколи обновлятися. В основному це переліки типів, видів, циклів

МОН – Міністерство освіти і науки України

МКР – модульна контрольна робота

ОКР – освітньо-кваліфікаційний рівень

ОКХ – освітньо-кваліфікаційні характеристики

ОПП – освітньо-професійні програми

НП – навчальний план

РНП – робочий навчальний план

КР – курсова робота

СРС – самостійна робота студентів

КП – курсовий проект

РГР – розрахунково-графічна робота

ДКР – домашня контрольна робота

СЛС – структурно-логічна схема

НТУУ «КПІ» – Національний технічний університет України «Київський політехнічний інститут»

AJAX – Asynchronous JavaScript And XML - підхід до побудови користувацьких веб інтерфейсів

ECTS – European Credit Transfer and Accumulation System (Європейська система переведення і накопичення кредитів) ANSI – American National Standards Institute (Американський національний інститут стандартів)

XML – eXtensible Markup Language (розширювана мова розмітки)

TMS – tenant management service – система управління тенантами

TCR – tenant connection resolver – бібліотека по управлінню тенантами

JRE – Java Runtime Environment – Набір необхідних утиліт для запуску скомпільованої програми, написаної на Java

JDK – Java Development Kit – Набір необхідних утиліт для розробки програм на мові програмування Java

URL – Uniform Resource Locator – система уніфікованих адрес електронних ресурсів, або однотипний визначник місцезнаходження ресурсу

HTML – HyperText Markup Language – стандартизована мова розмітки документів у Всесвітній павутині

IDE – Integrated Development Environment – система програмних засобів, яка використовується програмістами для розробки програмного забезпечення

TENANT – клієнт, арендатор – клієнт(замовник), який використовує програмний продукт

ВСТУП

Кожен розробник програмного забезпечення має за мету привернення якомога більшої кількості клієнтів. Ця ціль створює необхідність обслуговування та підтримки кожного з них.

Існує декілька підходів для підтримки багатоклієнтності. Один з них – створювати та деплоїти по одному (інколи і більше) екземпляру продукту для кожного клієнта. Проте, такий підхід досить затратний, адже потребує коштовних обчислювальних ресурсів комп'ютера. На противагу йому приходить рішення використовувати один екземпляр програмного забезпечення, який може обслуговувати багатьох клієнтів. У цьому випадку потрібно ідентифікувати клієнта та обслуговувати ресурси відповідно. Наприклад, у кожного клієнта є своя база даних, або інтеграція зі стороннім ресурсом за персональними повноваженнями. Одним із недоліків такого рішення є те, що необхідно додавати логіку по ідентифікації та обслуговуванню клієнта у сервісі з бізнес логікою. Це перечить одному з принципів SOLID – Single Responsibility (Єдина відповідальність), адже мікросервіс окрім специфічної бізнес логіки мусить описувати логіку підтримки мультиклієнтності. Інший недолік – дублювання коду, адже таких сервісів може бути декілька, і кожен має описувати одну і ту сервісну логіку.

Метою даної роботи є розробка бібліотеки, яка буде займатись логікою підтримки багатьох клієнтів, а саме – ідентифікувати конкретний клієнт через заголовк HTTP запиту та забезпечувати з'єднання із відповідною базою даних. Це допоможе вирішити описані вище проблеми та прискорить розробку програмних продуктів, концентруючи увагу розробника на бізнес логіці продукту, а не на одноманітній роботі по ідентифікації та обслуговуванню різних підрозділів. На даний момент аналогічної бібліотеки немає. Кожен розробник прописує логіку по використанню методів мультиклієнтності з основною логікою продукту. Саме цей недолік має вирішити розроблена мною бібліотека.

Існує безліч галузей застосування даної бібліотеки: від маленьких продуктів, що підтримують декількох клієнтів до великих ентерпрайз аплікацій, що обслуговують

десятки клієнтів. У даній роботі ми використаємо бібліотеку для пітримки різних підрозділів університету одним і тим же програмним продуктом. Це дасть змогу розробляти програми для університету один раз, мати їх однотипними та покривати потреби не одного, а багатьох підрозділів, факультетів. При цьому кожен факультет матиме свою базу даних, буде самостійно підтримувати та обслуговувати дані.

1. СТВОРЕННЯ БІБЛІОТЕКИ ПІДТРИМКИ ОКРЕМИХ БАЗ ДАНИХ РІЗНИХ ПІДРОЗДІЛІВ УНІВЕРСИТЕТУ В МІКРОСЕРВІСНІЙ АРХІТЕКТУРІ

1.1. Призначення програмного забезпечення

Дане програмне забезпечення призначене для використання при побудуванні веб сервісів університету. Проте, основна бібліотека може використовуватись в будь-якій предметній області, вона не має привя’зки до конкретної доменної області.

Потенційі користувачі – архітектори та розробники програмного забезпечення університету, які дотримуються принципів повторного використання коду, а також прагнуть мінімізувати час, витрачений на рутинну роботу, сконцентрувавшись натомість на функціональних потребах продукту.

1.2. Задачі, які розв’язуються програмним забезпеченням

Дане програмне забезпечення має розв’язати наступні задачі:

- Ініціалізація та збереження інформації про підрозділи університету та відповідні їм бази даних;
- Аналіз HTTP запитів та ідентифікація конкретного підрозділу;
- Використання бази даних, що відповідає конкретному підрозділу;
- Швидка та зручна інтеграція з основними сервісами університету.

1.3. Засоби розробки

Для розробки програмного продукту використані наступні засоби:

- JDK 8
- Intellij IDEA Community Edition
- PostgreSQL 9.6
- pgAdmin 4
- Postman
- Google Chrome
- Maven
- Gradle

1.4. Опис компонентів

Програмний продукт складається з наступних компонентів: **Tenant Connection Resolver (TCR)** – основна бібліотека. Вона відповідає за ідентифікацію клієнта та забезпечення зєднання з необхідною базою даних. Бібліотека імпортується в основний проект за допомогою Maven та Spring Configuration.

Tenant Management Service (TMS) – сервіс, що відповідає за збереження та модифікацію інформації про різні підрозділи університету та бази даних, що їм належать.

User Service Facade – бекенд сервіс для управління студентами факультету. Він отримує HTTP запит, звертається до бази даних та повертає усіх студентів, що там збережені. Це основний сервіс, він імпортує та використовує TCR для підтримки різних підрозділів університету.

User Service Client – клієнт для управління студентами факультету. Містить фронтенд для відображення інформації про студентів університету. Кожен клієнт належить до певного підрозділу університету.

TMS DB – база даних, що містить інформацію про підрозділи університету та список баз даних, що їм належить.

User Service DB – база даних, що містить інформацію про студентів певного факультету.

1.5 Схема взаємодії компонентів системи

Базова модель взаємодії компонентів виглядає наступним чином:

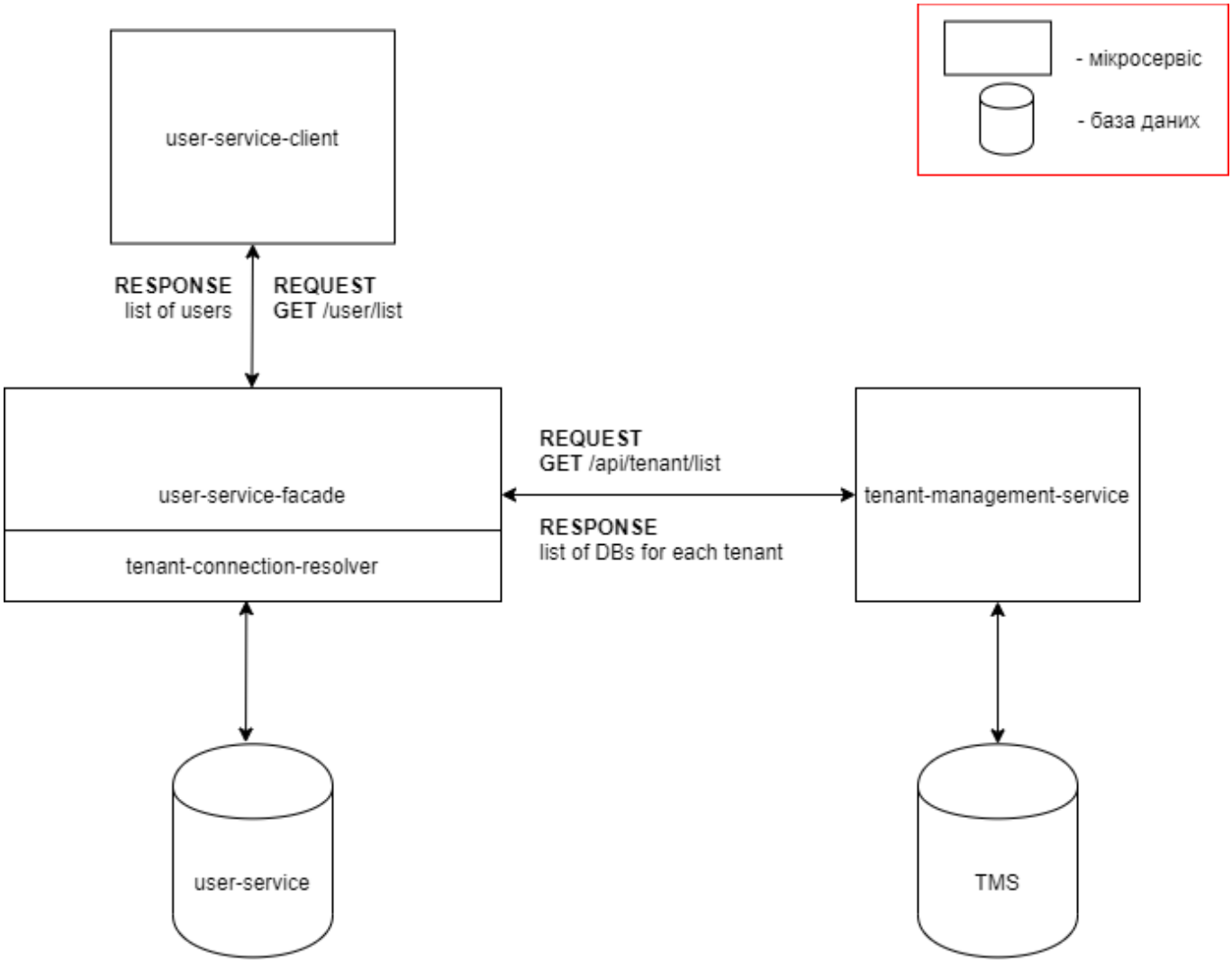

Рисунок 1.5.1 — Схема взаємодіїї компонентів системи

2. АНАЛІЗ ПРОБЛЕМИ ПІДТРИМКИ МУЛЬТИКЛІЄНТНОСТІ

Мультиклієнтність – властивість програмного забезпечення, що дозволяє одному й тому ж екземпляру програмного забезпечення задовольняти потреби двох чи більше клієнтів. Мультиклієнтність відрізняється від мультиекземплярності, тим що використовується лише один екземпляр програмного забезпечення, що обслуговує усіх клієнтів, а не по екземпляру на кожного клієнта.

2.1. Переваги мультиклієнтності

- **Збереження ресурсів**

Мультиклієнтність дозволяє заощадити витрати над масштабуванням за рахунок консолідації ресурсів IT в одну операцію. Екземпляр програми зазвичай вимагає певного обсягу пам'яті та обробних витрат, що може бути суттєвим при множенні на багато клієнтів, особливо якщо клієнти малі. Мультиклієнтність зменшує цю витрату, поширюючи програмне забезпечення на багатьох клієнтів. Подальша економія коштів може спричинитися за рахунок витрат на ліцензування базового програмного забезпечення (наприклад, операційних систем та систем управління базами даних). Якщо говорити грубо, якщо ви можете запустити все на одному екземплярі програмного забезпечення, вам потрібно придбати лише одну ліцензію на програмне забезпечення.

- **Збір та обробка даних**

Однією з найбільш переконливих причин використання мультиклієнтності є суттєві переваги для збору даних. Замість збирання даних з декількох джерел із потенційно різними схемами баз даних всі дані для всіх клієнтів зберігаються в єдиній схемі бази даних. Таким чином, виконувати запити серед клієнтів, видобувати дані та шукати тенденції набагато простіше.

- **Управління випусками**

Багатосторонність спрощує процес управління випуском. У традиційному процесі управління випуском пакети, що містять зміни коду та бази даних, поширюються на серверні машини клієнта; у випадку з одним примірником це буде одна серверна машина на кожного клієнта. Ці пакети потім мають бути встановлені на кожній окремій машині. За допомогою мультиклієнтної моделі пакет зазвичай повинен бути встановлений лише на одному сервері. Це значно спрощує процес управління випуском, і масштаб більше не залежить від кількості клієнтів.

2.2. Недоліки мультиклієнтності

- **Складність розробки**

Через додаткову складність налаштування та необхідність у підтримці метаданих для клієнта мультиклієнтні програми потребують більших зусиль розробки.

- **Складність масштабування клієнта**

Переваги мультиклієнтності можуть бути затемнені труднощами масштабування окремого екземпляра, оскільки попит зростає - підвищення продуктивності екземпляра на одному сервері може бути здійснено лише за рахунок придбання більш швидкого обладнання, таких як швидкі процесори, більше пам’яті та більш швидкі дискові системи, і зазвичай ці витрати зростають швидше, ніж якщо б навантаження було розділене між декількома серверами приблизно з однаковою сукупною ємністю.

2.3. Способи підтримки мультиклієнтості

2.3.1. Окрема база даних для кожного клієнта

Кожен клієнт має свою власну базу даних та ізольований від інших клієнтів.

Переваги полягають у тому, що бази даних незалежні і клієнти не можуть негативно впливати один на одного. Також наванатаження на базу не може бути більшою, ніж активність конкретного клієнта.

Основні недоліки – потрібно конфігурувати бази даних під кожного клієнта. Потрібно відкривати та підтримувати більше з’єднань з різними базами даних у програмі.

2.3.2. Спільна база даних та спільні схеми і таблиці

Усі клієнти мають спільну базу даних та спільні схеми і таблиці. Кожна таблиця містить окрему колонку, яка вказує, якому клієнту належить запис.

Переваги полягають у тому, що потрібно конфігурувати та підтримувати лише одну базу даних.

Основні недоліки – навантаження на базу зростає пропорційно кількості клієнтів. Особливо це відчутно при використанні транзакцій, індексів та інших інструментів баз даних. Існує ризик впливу клієнтами на дані один одного. Необхідно зберігати метадані разом з бізнес об’єктами. Завершення підтримки клієнта ускладнена, адже необхідно вичищати дані з існуючої активнох бази.

2.3.3 Спільна база даних але різні схеми та таблиці

Усі клієнти мають спільну базу даних, але власні схеми та таблиці.

Переваги полягають у тому, що потрібно конфігурувати та підтримувати лише одну базу даних. Доступ до даних обмежується схемою, тобто доступ клієнтів до своїх даних не залежатиме від інших клієнтів.

Основні недоліки – навантаження на базу зростає пропорційно кількості клієнтів. Особливо це відчутно при використанні транзакцій, індексів та інших інструментів баз даних.

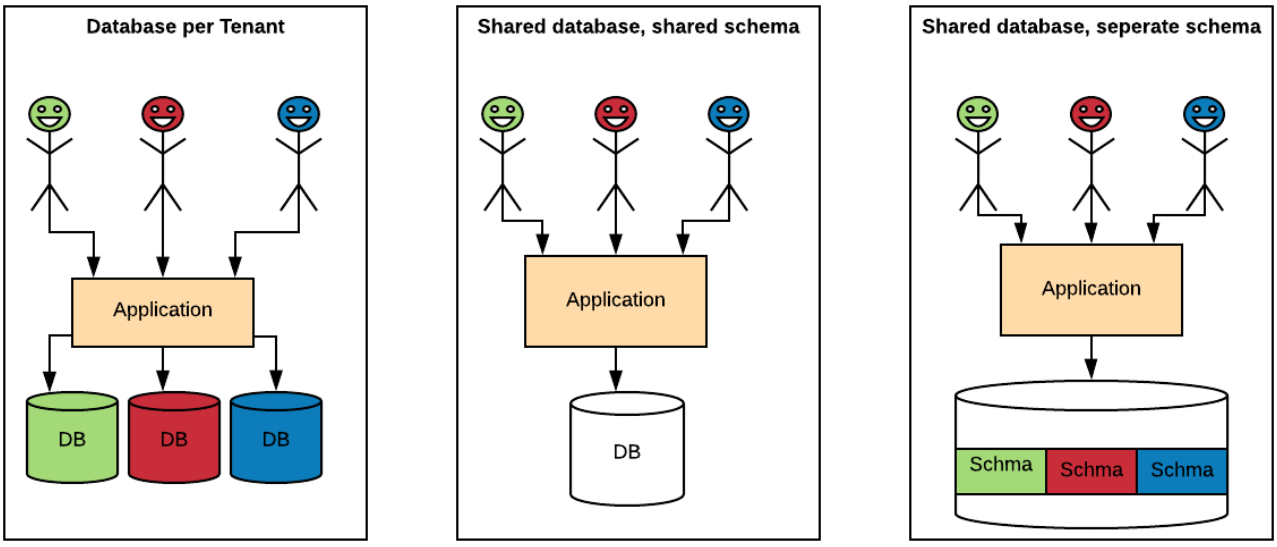

Рисунок 2.3.3.1 — Види мультиклієнтності

3. ЗАСОБИ РОЗРОБКИ

3.1. Операційна система

Розробка програмного забезпечення здійсювалась на операційній системі Windows 10 Pro. Але оскільки Java – мультиплатформна мова програмування, то розробка та використання може здійснюватись на будь-якій операційній системі, куди можна встановити Java Development Kit та Java Runtime Environment.

3.2. Java Development Kit

Розробка програмного забезпечення здійсювалась на мові програмування Java.

Для цього впершу чергу необхідно встановити Java Development Kit (JDK). Я використав JDK 1.8.0_241. Ця версія пропонує багато зручних засобів, такі як лямбда-вирази, функціональні інтерфейси та інші.

3.3. PostgreSQL 9.6

У якості СУБД я використав PostgreSQL 9.6. PostgreSQL, також відомий як Postgres, це безкоштовна система управління реляційними базами даних (СУРБД), з відкритим кодом. PostgreSQL імплементує операції з властивостями Atomicity, Consistency, Isolation, Durability (ACID), автоматично оновлюються представлення, матеріалізовані подання, тригери, зовнішні ключі, і збережені процедури. Він призначений для роботи з різними навантаженнями, від окремих машин до сховищ даних або веб-сервісів з багатьма одночасними користувачами. Це база даних за замовчуванням для сервера macOS, а також доступна для Linux, FreeBSD, OpenBSD та Windows.

3.4. Intellij IDEA

У якості середовища розробки я використовув Intellij IDEA Ultimate 2019.3. Це досить зручна та функціональна програма, яка має безліч зручних функцій та велику базу плагінів.

3.5. PgAdmin4

Для зручної роботи з СУБД я використовував програму PgAdmin4.

3.6. Postman

Для зручного тестування ендпоінтів я використовував Postman 7.2.

3.7. Google chrome

Для зручного тестування клієнтської частини я використовував браузер Google Chrome.

3.8. Maven 3.6.3 та Gradle 6.4

Для швидкої збірки проектів я використовую Maven 3.6.3 та Gradle 6.4.

4. ОПИС ПРОГРАМНОЇ РЕАЛІЗАЦІЇ

Для демонстрації роботи бібліотеки я створив два клієнта, що представляють факультети TEF та WELD. Тобто ми маємо два інстанса user-service-client. Також ми маємо сервіс user-service-facade, що використовує бібліотеку tenant-connection-resolver та сервіс tenant-management-service. Сервіс tenant-management-service використовує базу tms. Сервіс user-service-facade використовує бази user-service TEF та user-service WELD. Конкретна база залежить від того, від кого приходить запит - user-service-client TEF чи user-service-client WELD.

Загальна схема взаємодіїї сервісів та баз даних виглядатиме наступним чином (рисунок 4.1).

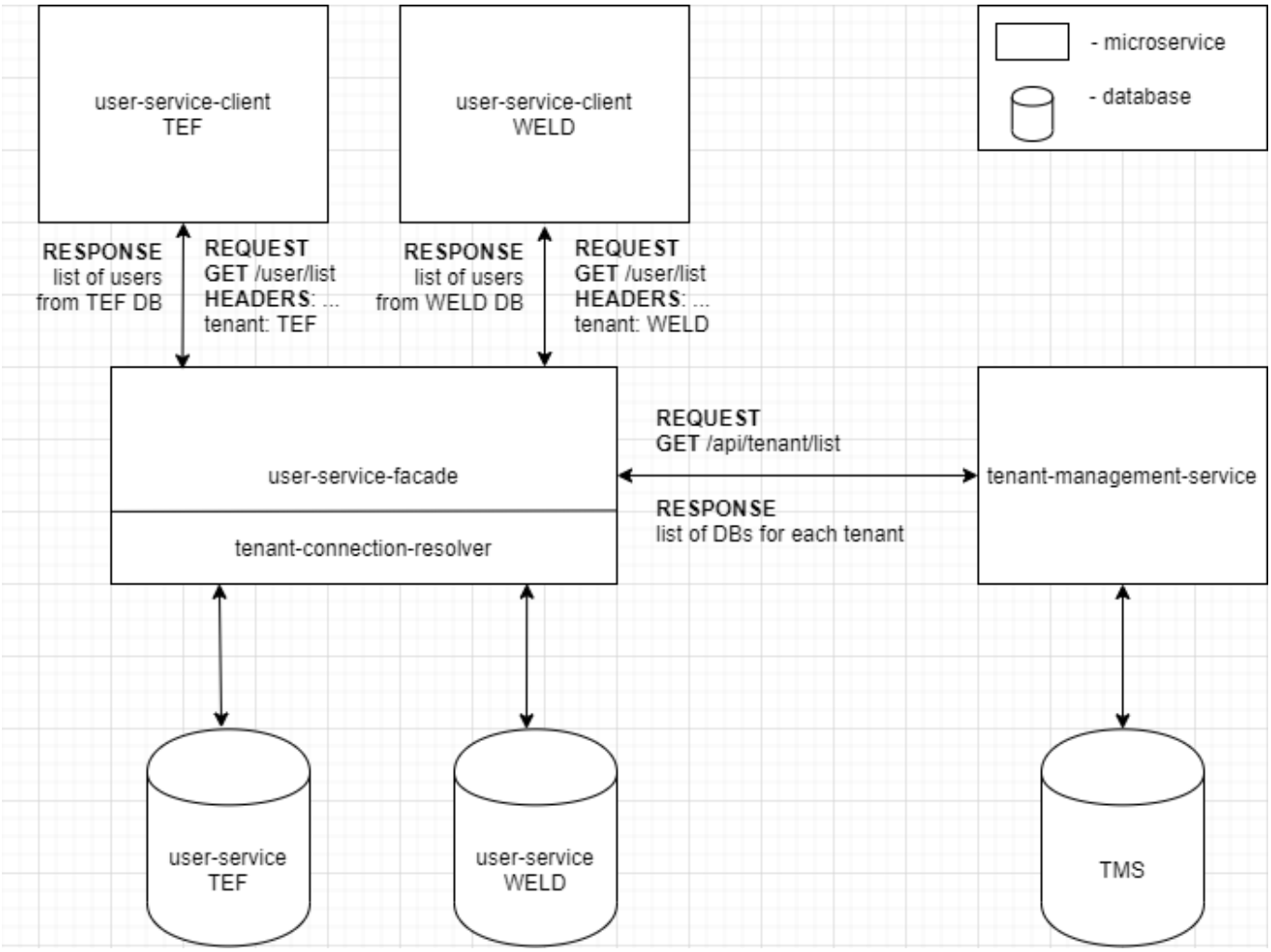

Рисунок 4.1 — Схема взаємодії сервісів та баз даних

4.1. Бібліотека tenant-connection-resolver

Tenant-Connection-Resolver – основна бібліотека, що містить логіку по ідентифікації клієнта, збереженню клієнта у контексті потоку, а також резолюції з’єднання з базою даних цього конкретного клієнта. Бібліотека написана на Java і збудована на технології Spring Boot Starter. Ця технологія дає можливість швидко і зручно імпортувати дану бібліотеку у будь-який Spring Boot проект, зконфігурувати необхідні властивості та використовувати бібліотеку у програмі. При цьому проект залишається не залежним від бібліотеки, адже кількість коду, необхідного для роботи з бібліотекою складає лише один рядок підключення конфігурації.

4.1.1 Структура проекту

Проект складається із таких основних пакетів (рисунок 4.1.1):

- Configuration – Spring Boot конфігурація (створення необхідних бінів);
- Context – класи, пов'язані з контекстом потоку;
- Domain – доменні класи проекту;
- Tms - класи для ініціалізації списку баз даних для клієнтів за допомогою tenant management service;
- Converter – класи для конвертації даних з ендпоінту ініціалізації в доменну модель проекту;
- Routing – класи для пошуку потрібної бази даних;
- Web – класи для перехоплення Http запиту, ідентифікації клієнта та збереження його в контекст потоку.

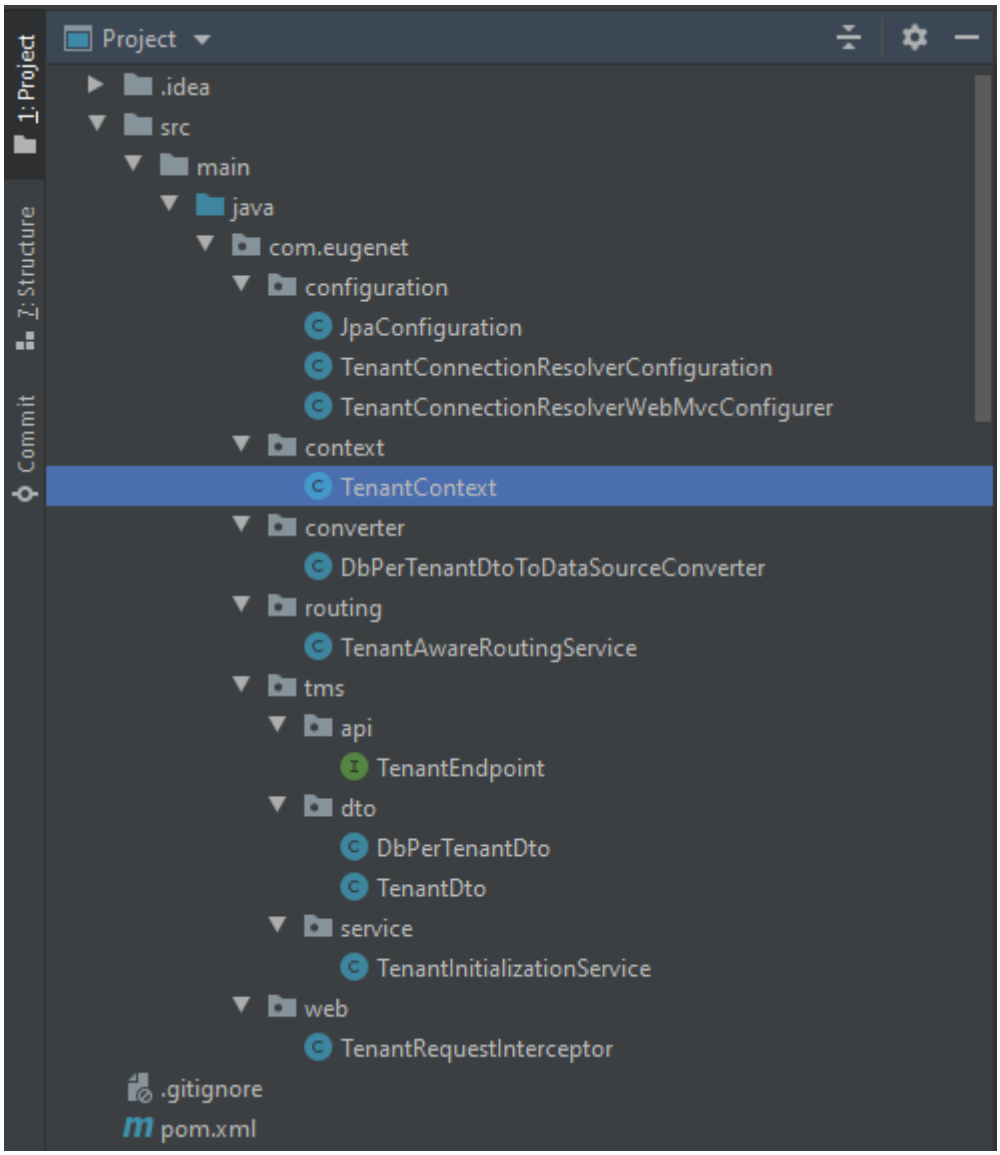

Рисунок 4.1.1 — Структура проекту tenant-connection-resolver

Проект складається із таких основних класів:

- TenantRequestInterceptor – перехоплює HTTP запит, вилучає значення заголовку з певним іменем. Ім’я заголовку конфігурується у сервісі, що використовує бібліотеку, за шляхом: ${tenant-connection-resolver.tenant-header}. Вилучене значення заголовка зберігається у TenantContext для активного потоку. Після виконання HTTP запиту значення заголовка вичищається із TenantContext. Відповідні методи preHandle та afterCompletion викликає Spring Container. Це відбувається завдяки тому, що клас розширяє HandlerInterceptorAdapter (таблиця 4.1.1).

Таблиця 4.1.1 – TenantRequestInterceptor

public class TenantRequestInterceptor extends HandlerInterceptorAdapter

- TenantContext – використовується як сховище даних для конкретного потоку (таблиця) 4.1.2. Використовує клас ThreadLocal для забезпечення видимості даних лише у рамках того потоку, в якому вони були ініціалізовані. Містить методи для ініціалізації, отримання та видалення даних.

Таблиця 4.1.2 – TenantContext

public class TenantContext {

- TenantAwareRoutingService – відповідає за забезпечення з’єднання з необхідною базою даних. Розширює клас AbstractRoutingDataSource, що реалізований бібліотекою Spring JDBC (таблиця 4.1.3). Основна логіка описана саме в ньому, у класі-спадкоємцю ми лише описуємо логіку по пошуку ключа для бази. Ключем виступає значення клієнта, що зберігається у класі TenantContext.

Таблиця 4.1.3 - TenantAwareRoutingService

public class TenantAwareRoutingService extends AbstractRoutingDataSource

- TenantInitializationService – відповідає за ініціалізацію мапи з’єднань за базою даних по клієнту (таблиця 4.1.4). Звертається до сервісу tenant-management-service через HTTP запит та отримує у відповідь список усіх актуальних клієнтів з їх активними базами даних. Даний клас містить параметер ${tenant-connection-resolver.db-name}. Цей параметер ініціалізується у сервісі, що використовує бібліотеку та вказує, яку базу даних клієнта треба зберегти та використовувати для поточного сервісу. Запит до tenant-management-service відбувається за допомогою класу TenantEndpoint.

Таблиця 4.1.4 - TenantInitializationService

```
@RequiredArgsConstructor
public class TenantInitializationService {
```

- TenantEndpoint – відповідає за з’єднання з tenant-management-service. Цей клас збудований на технології Feign, виступає компонентом FeignClient (таблиця 4.1.5). Використовує параметр ${tenant-connection-resolver.url}. Цей параметер ініціалізується у сервісі, що використовує бібліотеку та вказує, за якою адресою знаходиться tenant-management-service.

Таблиця 4.1.5 - TenantEndpoint

```
@FeignClient(name = "TenantEndpoint", url = "${tenant-connection-
resolver.url}")
public interface TenantEndpoint {
```

- TenantDto – містить інформацію про клієнта (таблиця 4.1.6). Містить наступні поля:
    - tenantId – унікальний ідентифікатор;
    - tenantAlias – аліас, друге ім’я;
    - description – опис;
    - active – активність клієнта;
    - dbPerTenantsDto – список баз даних цього клієнта.

Таблиця 4.1.6 - TenantDto

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class TenantDto {
```

- DbPerTenantDto – містить інформацію про базу даних клієнта (таблиця 4.1.8). Містить наступні поля:
  - url – адреса бази даних;
  - username – ім’я користувача;
  - userPassword – пароль користувача;
  - driverClassName – назва класу драйвера для з’днання;
  - active – активність бази даних;

Таблиця 4.1.8 - DbPerTenantDto

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class DbPerTenantDto {
```

- DbPerTenantDtoToDataSourceConverter – конвертує об’єкт класу DbPerTenantDto в об’єкт класу DataSource (таблиця 4.1.9). DbPerTenantDto та TenantDto потрібні для конвертації сирих даних HTTP відповіді у модель об’єкту. DataSource використовується Spring Jpa для взаємодії з базою даних.

Таблиця 4.1.9 – DbPerTenantDtoToDataSourceConverter

```
public class DbPerTenantDtoToDataSourceConverter {
```

- JpaConfiguration – Spring конфігурація, що описує бін DataSource (таблиця 4.1.10). У нашому випадку ми повертаємо не якийсь конкретний об’єкт бази даних, а об’єкт класу TenantAwareRoutingService. Таким чином ми впроваджуємо логіку мультиклієнтості у бібліотеці на рівень бази даних. Під час ініціалізації клас звертається до об’єкту класу TenantInitializationService для отримання списку клієнтів та їх баз даних. Після цього використовується об’єкт класу DbPerTenantDtoToDataSourceConverter для конвертаціїї об’єктів класу DbPerTenantDto у об’єкти класу DataSource.

Таблиця 4.1.10 – JpaConfiguration

```
@RequiredArgsConstructor
public class JpaConfiguration {
```

- TenantConnectionResolverWebMvcConfigurer – Spring конфігурація для реєстрації об’єкту класу TenantRequestInterceptor (таблиця 4.1.11). Без цієї реєстрації веб фільтр не буде виконуватись, тобто не буде процесу ініціалізації клієнта з HTTP запиту.

Таблиця 4.1.11 – TenantConnectionResolverWebMvcConfigurer

```
@Configuration
@RequiredArgsConstructor
public class TenantConnectionResolverWebMvcConfigurer implements
WebMvcConfigurer {
```

- TenantConnectionResolverConfiguration – Spring конфігурація для створення та ініціалізації об’єктів класів, що описані у бібліотеці (таблиця 4.1.12). Створює

об’єкти класів TenantRequestInterceptor, TenantInitializationService, DbPerTenantDtoToDataSourceConverter. Також імпортує інші конфігурації – JpaConfiguration та TenantConnectionResolverWebMvcConfigurer. Таким чином, для використання бібліотеки у інших сервісах необхідно імпортувати лише даний клас, і усі об’єкти бібліотеки будуть доступні та будуть функціонувати у ньому. Також даний клас містить код, що дозволяє бібліотеці створити Feign Client над класом TenantEndpoint.

Таблиця 4.1.12 – TenantConnectionResolverWebMvcConfigurer

```
@Configuration
@EnableFeignClients(clients = TenantEndpoint.class)
@Import(value = {JpaConfiguration.class,
TenantConnectionResolverWebMvcConfigurer.class})
public class TenantConnectionResolverConfiguration {
```

Проект містить такі основні залежності:

- org.springframework.boot:spring-boot-starter-actuator:1.5.9;
- org.springframework.boot:spring-boot-starter-web:1.5.9;
- org.springframework.boot:spring-boot-starter-data-jpa:1.5.9;
- org.springframework.boot:spring-cloud-starter-feign:1.5.9;
- com.google.guava:guava:21.0;
- org.apache.commons:commons-lang3: 3.10;
- org.projectlombok:lombok: 1.18.2;

Проект потребує декларації таких змінних:

- ${tenant-connection-resolver.url};
- ${tenant-connection-resolver.tenant-header};
- ${tenant-connection-resolver.tenant-header};

4.2. Сервіс user-service-facade

User-Service-Facade – Java проект, на якому ми продемонструємо роботу бібліотеки tenant-connection-resolver. Даний проект описує бекенд сервіс, який повертає список користувачів, що збережені у певній базі даних. Бібліотека збудована на технології Spring Boot. Ця технологія дає можливість швидко і зручно підключати будь-які залежності, Spring Boot стартери, збирати та деплоїти даний сервіс.

4.2.1 Структура проекту

Проект складається із таких основних пакетів (рисунок 4.2.1):

- Config – Spring Boot конфігурація (створення необхідних бінів);
- Controller – класи – контролери згідно з шаблоном MVC; Збудовані на технології Spring Web.
- Service – класи – сервіси, що містять бізнес логіку;
- Entity – класи – репрезентують об’єкти, що зберігаються у базі даних
- Repository – класи репозиторіїї для доступу в базу даних. Збудовані на технології Spring Data JPA.

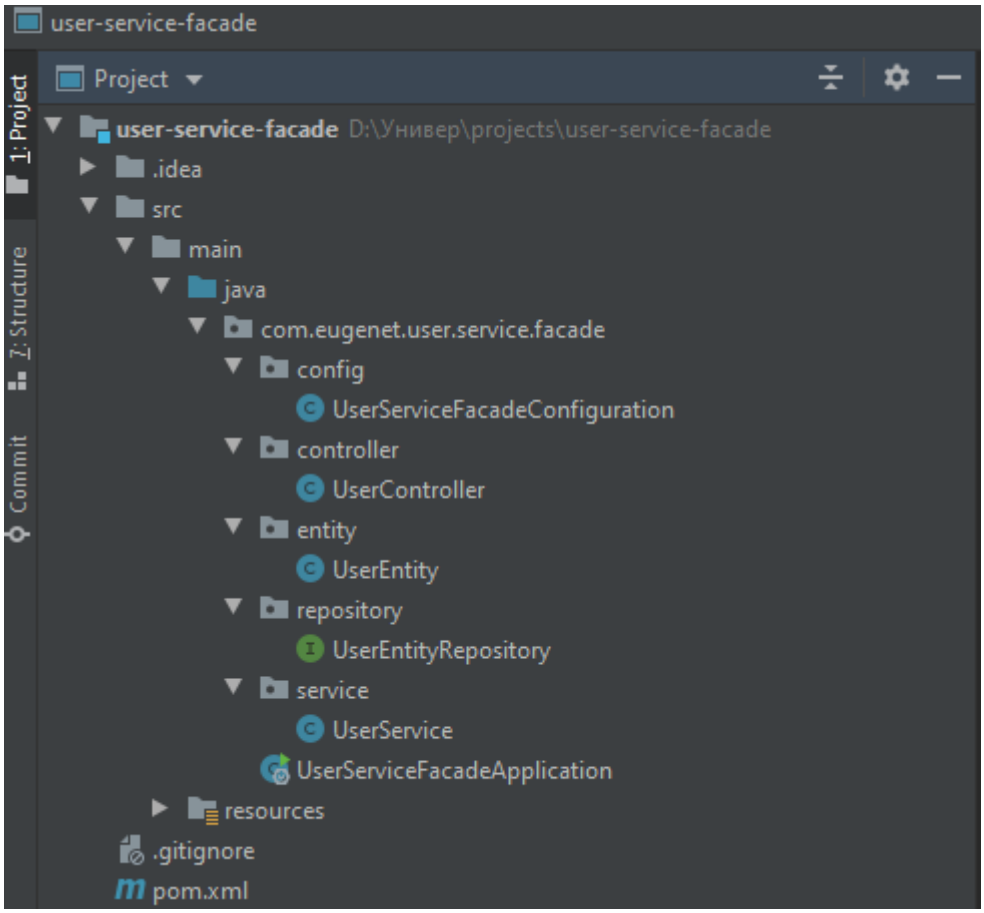

Рисунок 4.2.1 — Структура проекту user-service-facade

Проект складається із таких основних класів:

- UserController – Spring Controller (таблиця 4.2.1). Виступає як вхідна точка доступу до даних. Містить метод, що повертає список усіх користувачів певною системою. Метод відпрацює на GET запит за адресою http://{application-path}/user/list, де {application-path} – шлях до даного сервісу, приклад: localhost:8080. Контролер делегує запит до об’єкту-сервісу класу UserService. Хочу звернути увагу на те, що під кожен HTTP запит до контролера буде виділено окремий поток виконання. Саме це дає змогу використовувати TenantContext та TenantRequestInterceptor і бути впевненим у тому, що на будь-якому проміжку часу від ініціалізації до очищення потоку буде повернуто лише актуальне значення активного клієнта. При початку HTTP запиту буде збережно значення клієнта, після виконання перед завершенням це значення буде видалено з контексту.

Таблиця 4.2.1 - UserController

```
@Controller
@RequestMapping("/user")
@RequiredArgsConstructor
public class UserController {
```

- UserService – сервіс для роботи з користувачами (таблиця 4.2.2). Містить метод, що повертає усіх користувачів певної системи. Даний сервіс делегує запит до об’єкту класу UserRepository.

Таблиця 4.2.2 - UserService

```
@Service
@RequiredArgsConstructor
public class UserService {
```

- UserEntityRepository – репозиторій, клас для роботи з базою даних. Збудований на технології Spring Data JPA (таблиця 4.2.3). Даний інтерфейс розширює інтерфейс JpaRepository. У описані основні методи ОРМ взаємодії з базою даних. Використання анотаціїї @Repository вказує Spring контейнеру, що необхідно створити об’єкт класу, що реалізує дані методи. Нижче цього шару абстракції лежить взаємодія з об’єктами класу DataSource для отримання з’єднання з базою та виконання необхідної роботи. Саме тут використовується об’єкт класу TenantAwareRoutingService, що повертає різні об’єкти DataSource для різних клієнтів.

Таблиця 4.2.3 – UserEntityRepository

```
@Repository
public interface UserEntityRepository extends JpaRepository<UserEntity, Long> {
```

- UserServiceFacadeConfiguration – Spring конфігурація для імпорту TenantConnectionResolverConfiguration із бібліотеки tenant-connection-resolver (таблиця 4.2.4). Таким чином, при додаванні єдиного імпорту ми отримуємо усі об’єкти бібліотеки tenant-connection-resolver.

Таблиця 4.2.4 – UserServiceFacadeConfiguration

```
@Configuration
@Import(TenantConnectionResolverConfiguration.class)
public class UserServiceFacadeConfiguration {
```

- UserServiceFacadeApplication – клас для опису Spring Boot Application (таблиця 4.2.5). Це основна точка входу для створення та деплою сервісу фреймворком.

Таблиця 4.2.5 – UserServiceFacadeConfiguration

```
@SpringBootApplication
public class UserServiceFacadeApplication {

    public static void main(String[] args) {
        SpringApplication.run(UserServiceFacadeApplication.class, args);
    }
}
```

Проект має наступні параметри:

- tenant-connection-resolver.tenant-header: tenant – ім’я заголовку, в якому передається значення клієнта;
- tenant-connection-resolver.db-name: user-service – ім’я бази даних, в якій збергіаються значення користувачів;

- tenant-connection-resolver.url: localhost:8081/api/tenant/list – адреса tenant-management-service;
- server.port: 8082 – адреса порту, по якому буде доступ до даного сервісу;
- spring.datasource.initialize: false – вказує на те, що ініціалізація з’єднання з базою даних має бути lazy, тобто при першому виклику. У іншому випадку ініціалізація відбувається одразу при створенні об’єкту DataSource;
- spring.jpa.open-in-view – вказує на те, що необхідно створювати інстанс класу EntityManager під кожний поток, тобто під кожний HTTP запит;
- spring.jpa.show-sql: true – вказує на необхідність виводити виконані sql запити у консоль;
- spring.jpa.database: postgresql – вказує на те, яка база даних використовується даним сервісом;

4.3. Сервіс user-service-client

User-Service-Client – Java проект, який виступатиме у ролі клієнта для user-service-facade. Даний проект описує веб додаток, що містить фронтенд та бекенд. На фронтенді можливо зайти та подивитися список студентів певного факультету. Бекенд відповідає за отримання та опрацювання даних. Цей сервіс не містить бази даних, а звертається до user-service-facade, запитуючи список користувачів та вказуючи, якому факультетові належить конкретний сервіс. Таким чином, інстансів даного сервісу може бути безліч, і вони можуть покривати потреби багатьох різних факультетів певного університету.

4.3.1 Структура проекту

Проект складається із таких основних пакетів (рисунок 4.3.1):

- api – містить класи – ендпоінти для надсилання HTTP запитів до user-service-facade та отримання відповідей;

- controller – містить класи – контролери, для отримання та опрацювання запитів від frontend;
- domain – містить класи, що описують бізнес – сутності;
- service – містить класи, що відповідають за бізнес логіку клієнта;
- templates – містить Thymeleaf темплейти для конвертації в UI шар.

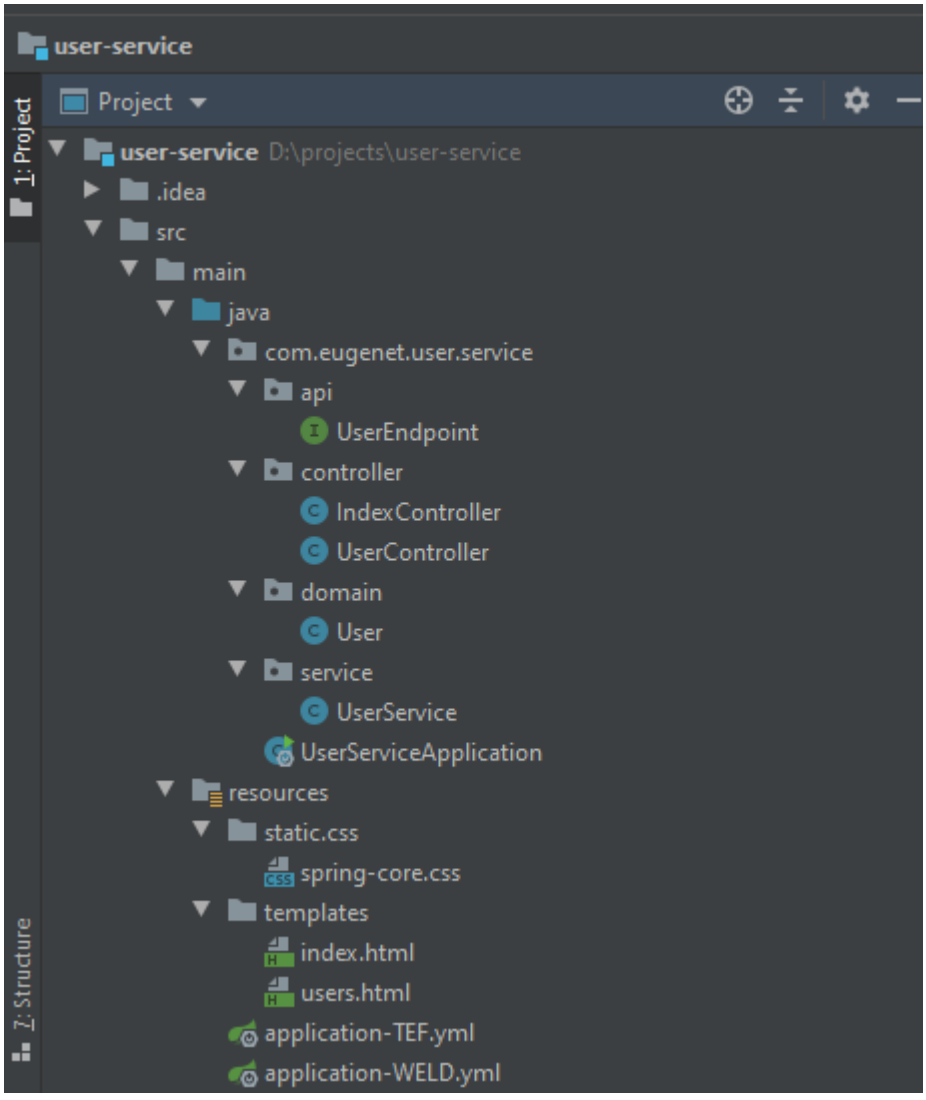

Рисунок 4.3.1 — Структура проекту user-service-client

Проект складається із таких основних класів:

- IndexController – Spring Controller (таблиця 4.3.1). Метод відпрацює на GET запит за адресою http://{application-path, де {application-path} – шлях до даного сервісу, приклад: localhost:8080. Тобто контролер опрацьовує головну сторінку (індекс). Повертає на UI значення параметру ${user-service.tenant}, що вказує на те, який клієнт активний для даного сервісу.

Таблиця 4.3.1 – IndexController

```
@Controller
public class IndexController {
```

- UserController – Spring Controller (таблиця 4.3.2). Виступає як вхідна точка доступу до даних. Опрацьовує запити від UI. Містить метод, що повертає список усіх користувачів певною системою. Метод відпрацює на GET запит за адресою http://{application-path}/user/list, де {application-path} – шлях до даного сервісу, приклад: localhost:8080. Контролер делегує запит до об’єкту-сервісу класу UserService.

Таблиця 4.3.2 – UserController

```
@Controller
@RequestMapping("/user")
@RequiredArgsConstructor
public class UserController {
```

- UserService – сервіс для роботи з користувачами (таблиця 4.3.3). Містить метод, що повертає усіх користувачів певної системи. Даний сервіс делегує запит до об’єкту класу UserEndpoint. Він використовує та передає значення параметру ${user-service.tenant}, вказуючи, для якого клієнта необхідно запросити дані.

Таблиця 4.3.3 – UserService

```
@Service
@RequiredArgsConstructor
public class UserService {
```

- UserEndpoint – відповідає за з’єднання з user-service-facade. Цей клас збудований на технології Feign, виступає компонентом FeignClient (таблиця 4.3.4). Використовує параметр ${user-service.facade-url}. Цей параметер вказує, за якою адресою знаходиться user-service-facade.

Таблиця 4.3.4 – UserEndpoint

```
@FeignClient(name = "user-endpoint", url = "${user-service.facade-url}/user")
public interface UserEndpoint {
```

- User – клас, що репрезентує модель даних, які використовуються сервісом (таблиця 4.3.5). Містить наступні поля:
  - id – унікальний ідентифікатор;
  - firstName – ім’я;
  - lastName – прізвище;
  - email – електронна адреса;
  - department – факультет, до якого належить студент;
  - averagePoint – середня оцінка;

Таблиця 4.3.5 – User

```
@Getter
@Setter
public class User {
```

- UserServiceApplication – клас для опису Spring Boot Application (таблиця 4.3.6). Це основна точка входу для створення та деплою сервісу фреймворком.

Таблиця 4.3.6 – UserServiceApplication

```
@EnableFeignClients
@SpringBootApplication
public class UserServiceApplication {

    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
}
```

Проект використовує технологію Thymeleaf для генерації HTML сторінок із шаблону. Основні веб сторінки:

- index.html – шаблон для відображення головної сторінки проекту;
- users.html – шаблон для сторінки, що відображає список студентів з даними про них;

Проект має наступні параметри:

- user-service.tenant: WELD / TEF – назва поточного клієнта (факультета);
- user-service.facade-url: localhost:8082 – адреса user-service-facade;
- server.port: 8083 / 8084 – адреса порту, по якому буде доступ до даного сервісу;

4.4. Детальна схема взаємодії компонентів

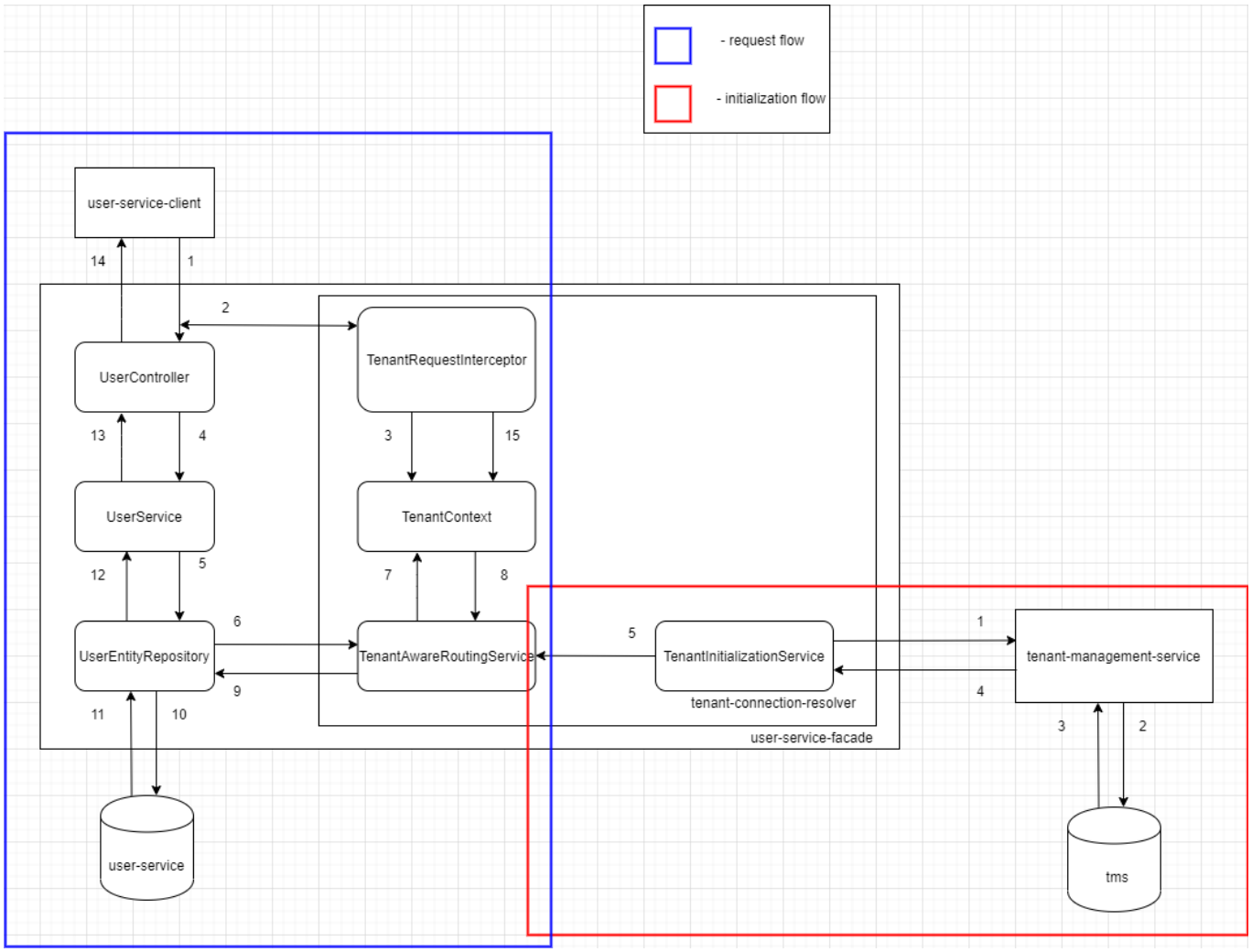

Рисунок 4.4 — Детальна схема взаємодії класів сервісів та класів

Ми будемо використовувати наступні скорочення, щоб краще зрозуміти, який сервіс чи бібліотека виконує поточний крок:

- [usc] – user-service-client - сервіс
- [usf] – user-service-facade - сервіс
- [tcr] – tenant-connection-resolver - бібліотека
- [tms] – tenant-management-service – сервіс
- [tms-db] – база даних tms
- [us-db] – база даних user-service

4.5. Покроковий опис процесу ініціалізації

1) [tcr] TenantInitializationService надсилає HTTP запит в [tms] та просить повернути усіх активних клієнтів та відповідні їм бази даних.

2) [tms] звертається до бази даних та запитує усіх активних клієнтів та відповідні їм бази даних.

3) [tms-db] знаходить та повертає усіх активних клієнтів та відповідні їм бази даних.

4) [tms] конвертує усіх активних клієнтів та відповідні їм бази даних у HTTP відповідь та повертає в [usf].

5) [tcr] TenantInitializationService отримує усіх активних клієнтів та відповідні їм бази даних, вибирає лише бази, що відповідають патерну, заданому в файлі конфігурації (у нашому випадку – бази, що закінчуються на “user-service”). Викликає TenantAwareRoutingService та зберігає мапу, де ключ – ім’я клієнта, а значення – відповідна йому база даних.

4.6. Покроковий опис процесу HTTP запиту

1) [usc] на запит користувача в Google Chrome надсилає HTTP запит в user-service-facade. Запит містить header з ключем tenant, що містить ідентифікатор клієнта. Приклад: tenant: TEF.

2) [tcr] TenantRequestInterceptor перехоплює запит та отримує значення з хедеру tenant. У нашому прикладі це буде значення TEF.

3) [tcr] TenantRequestInterceptor передає значення поточного клієнта в TenantContext. TenantContext зберігає це значення для кожного потоку. Кожен HTTP запит виконується певним потоком. Тобто, значення зберігається до закінчення запиту. Також значення унікальне для кожного запиту.

4) [usf] TenantController отримує HTTP запит та викликає UserService.

5) [usf] UserService викликає UserEntityRepository.

6) [usf] UserEntityRepository викликає [tcr] TenantAwareRoutingService та запитує в нього DataSource – інстанс бази даних, в якій зберігаються користувачі для поточного клієнта.

7) [tcr] TenantAwareRoutingService викликає TenantContext та запитує в нього поточення значення клієнта.

8) [tcr] TenantContext видає значення, отримане та збережене з HTTP запиту.

9) [tcr]TenantAwareRoutingService знаходить відповідний DataSource для поточного клієнта та повертає його.

10) [usf] UserEntityRepository звертається до бази та запитує усіх користувачів.

11) [usdb] База даних знаходить усіх користувачів та повертає.

12) [usf] UserEntityRepository повертає користувачів в UserService.

13) [usf] UserService повертає користувачів в UserController.

14) [usf] UserController конвертує усіх користувачів у HTTP відповідь та повертає в [usc].

15) [tcr] TenantContext очищає поточне значення клієнта для цього потоку.

5. РОБОТА КОРИСТУВАЧА ІЗ СИСТЕМОЮ

Для успішного покрокового виконання необхідна ОС Windows.

- Встановимо JDK 8. Для цього необхідно зайти на офіційний веб сайт. Зареєструватись та скачати версію jdk-8u251-windows-x64.exe. Встановити скачану програму з налаштуваннями за замовчуванням.
- Встановимо Maven 3.6.3. Для цього необхідно зайти на офіційний веб сайт. Скачати файл apache-maven-3.6.3-bin.zip. Розпакувати архів та встановити програму.
- Встановимо Gradle 6.4. Для цього необхідно зайти на офіційний веб сайт. Скачати файл gradle-6.4.bin.zip. Розпакувати архів та встановити програму.
- Встановимо PostgreSQL 9.6. Для цього необхідно зайти на офіційний веб сайт. Скачати версію 9.6.17. Встановити скачану програму з налаштуваннями за замовчуванням. Для використання ПО без змін встановіть пароль користувача **postgres** як **password**.

5.1. Налаштування баз даних

Налаштуємо необхідні бази даних: tms, user-service:5432, user-service:5433. 5432 – порт, що використовується сервером postgres за замовчуванням. Бази tms та один з user-service будуть використовувати його. Проте, нам необхідна й друга база user-service для демонстрації мультиклієності. Тому нам необхідно буде створити ще один сервер на 5433 порті.

Натисніть Win+R, введіть cmd та натисніть Enter (рисунок 5.1.1).

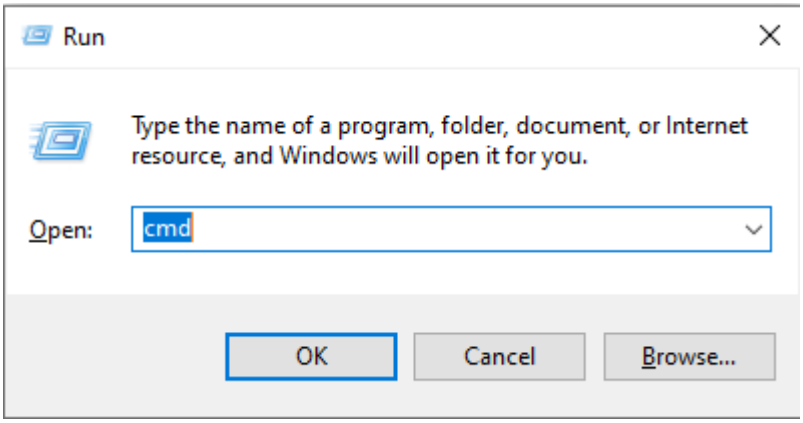

Рисунок 5.1.1 — Запуск командної строки

Введіть **psql -U postgres** та натисніть Enter. Після цього введіть запрошений пароль та натисніть Enter (рисунок 5.1.2).

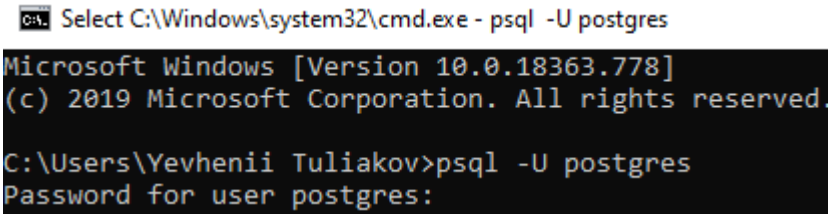

Рисунок 5.1.2 — Авторизація до серверу

З’єднання з сервером відкрито, далі можна створювати бази даних (рисунок 5.1.3).

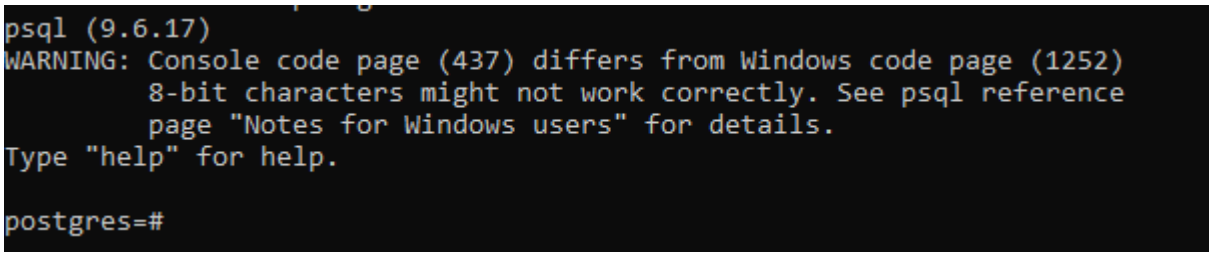

Рисунок 5.1.3 — З’єднання з сервером.

- Спочатку налаштуємо базу даних TMS.

Для початку виконаємо команду **CREATE DATABASE tms;** Ця команда створить саму базу (рисунок 5.1.4).

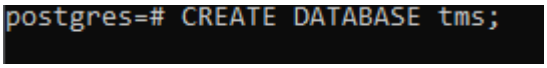

Рисунок 5.1.4 — Створення бази даних tms.

На даному етапі цього достатньо, оскільки таблиці будуть створенні за допомогою liquibase при старті сервісу tenant-management-service.

- Далі налаштуємо базу даних user-service:5432.

Для початку виконаємо команду **CREATE DATABASE** user-service**;** Ця команда створить саму базу (рисунок 5.1.5).

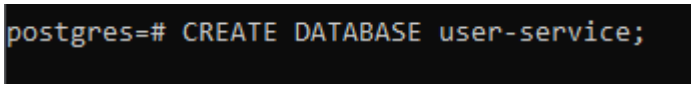

Рисунок 5.1.5 — Створення бази user-service.

Далі необхідно перейти до бази, виконавши команду **\c "user-service";** (рисунок 5.1.6).

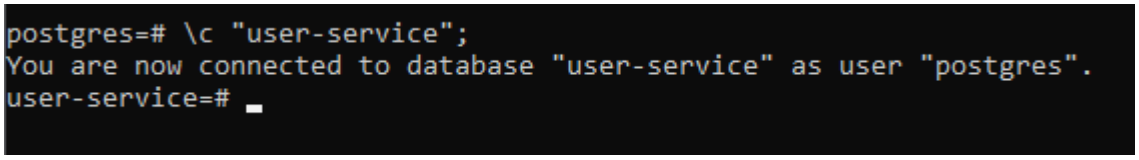

Рисунок 5.1.6 — Перехід до бази user-service.

Після цього необхідно створити таблицю, виконавши наступну команду (таблиця 5.1.1, рисунок 5.1.7).

Таблиця 5.1.1 – SQL команда для створення таблиці “user”

CREATE TABLE "user" (id SERIAL PRIMARY KEY, first_name VARCHAR(255), last_name VARCHAR(255), email VARCHAR(255), department VARCHAR(255), average_point VARCHAR(255));

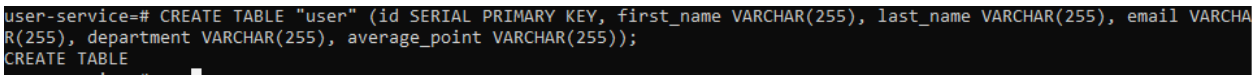

Рисунок 5.1.7 — Створення таблиці user.

Далі необхідно додати користувачів. Для нашого прикладу використаємо наступну команду. (таблиця 5.1.2, рисунок 5.1.8).

Таблиця 5.1.2 – SQL команда для заповнення таблиці “user”

insert into "user" values (1, 'Eugene', 'Korikh', 'eugene.korikh@mail.com', 'Department 1', '99.9'); та **insert into "user" values (2, 'Kate', 'Aksenova', 'kate.aksenova@mail.com', 'Department 1', '80');**

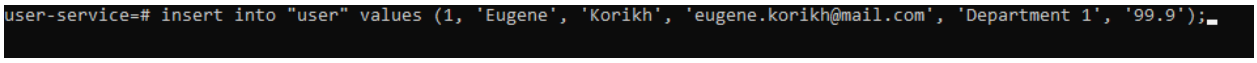

Рисунок 5.1.8 — Заповнення таблиці user.

- Далі налаштуємо базу даних user-service:5433.

Для початку необхідно створити папку для сервера та логів. Створимо папку **D:\postgres\db2** та створимо в ній іншу папку **log**. Далі запустимо командну строку та виконаємо команду (таблиця 5.1.3, рисунок 5.1.9**)**.

Таблиця 5.1.3 – Запуск серверу бази з портом 5433

pg_ctl -D D:/postgres/db2 -o "-p 5433" -l D:/postgres/db2/log start

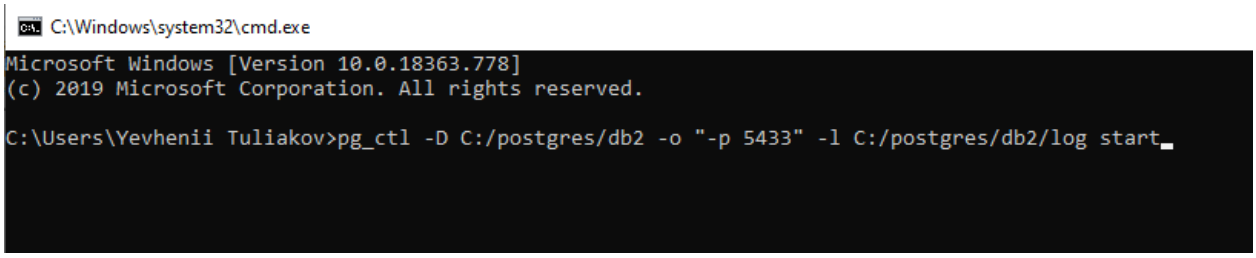

Рисунок 5.1.9 — Запуск серверу бази з портом 5433

Далі необхідно з’єднатися з сервером командою (таблиця 5.1.4).

Таблиця 5.1.4 – Запуск серверу бази з портом 5433

psql -p 5433 -U postgres

Після цього виконати всі кроки, аналогічно до попереднього пункту, змінивши лише дані студентів.

5.2. Налаштування tenant-management-service

У першу чергу необхідно створити папку, в яку ми будемо скачувати проекти. Я створив **D:/services**. Далі необхідно запустити командну строку та перейти до даної папки. Потім виконати команду **git clone https://gitlab.com/AMS-MICROSERVICES/tms** (рисунок 5.2.1).

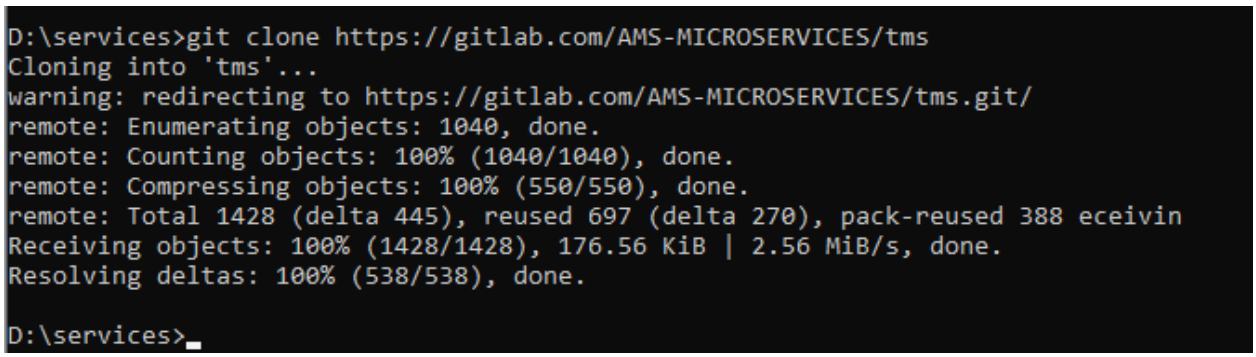

Рисунок 5.2.1 — Скачування проекту tms

Далі необхідно перейти в папку tms та виконати команду **git checkout TCR-integration** (рисунок 5.2.2).

```
D:\services>cd tms

D:\services\tms>git checkout TCR-integration
Switched to a new branch 'TCR-integration'
Branch 'TCR-integration' set up to track remote branch 'TCR-integration' from 'origin'.
```

Рисунок 5.2.2 — Зміна активної гілки на TCR-integration

Далі відкриємо проект в Intellij IDEA (рисунок 5.2.3 та 5.2.4).

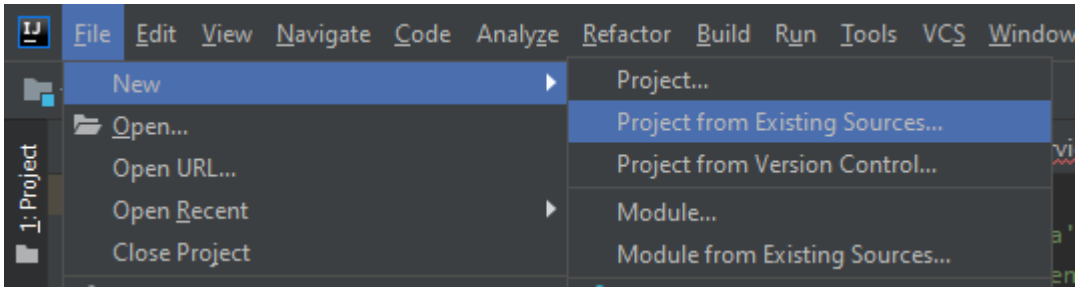

Рисунок 5.2.3 — Імпорт проекту в Intellij IDEA

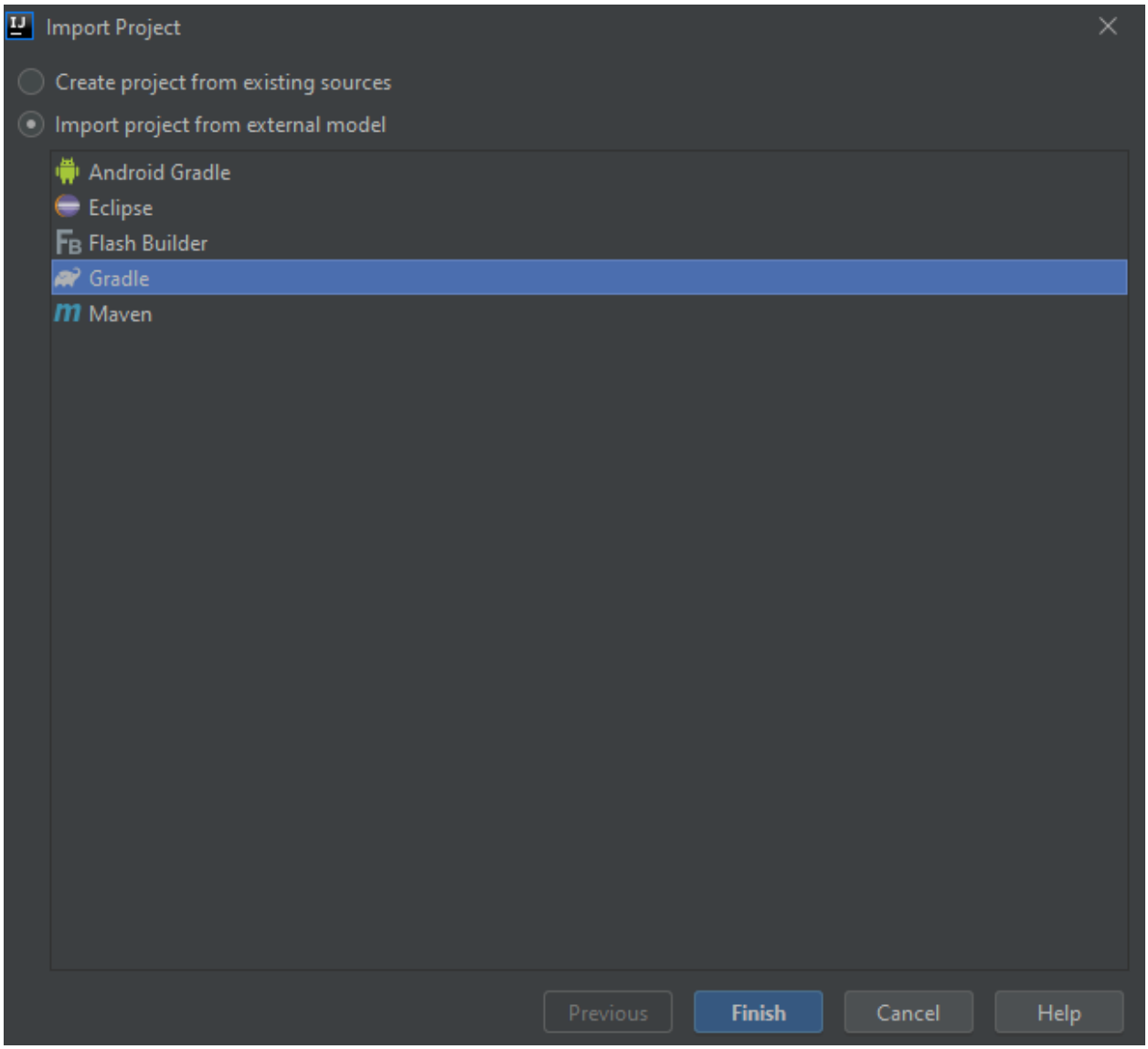

Рисунок 5.2.4 — Імпорт Gradle проекту в Intellij IDEA

Далі необхідно перейти до класу **TmsServiceApplication**, натиснути праву кнопку та вибрати пункт **Run ‘TmsServiceApplication’** (рисунок 5.2.5).

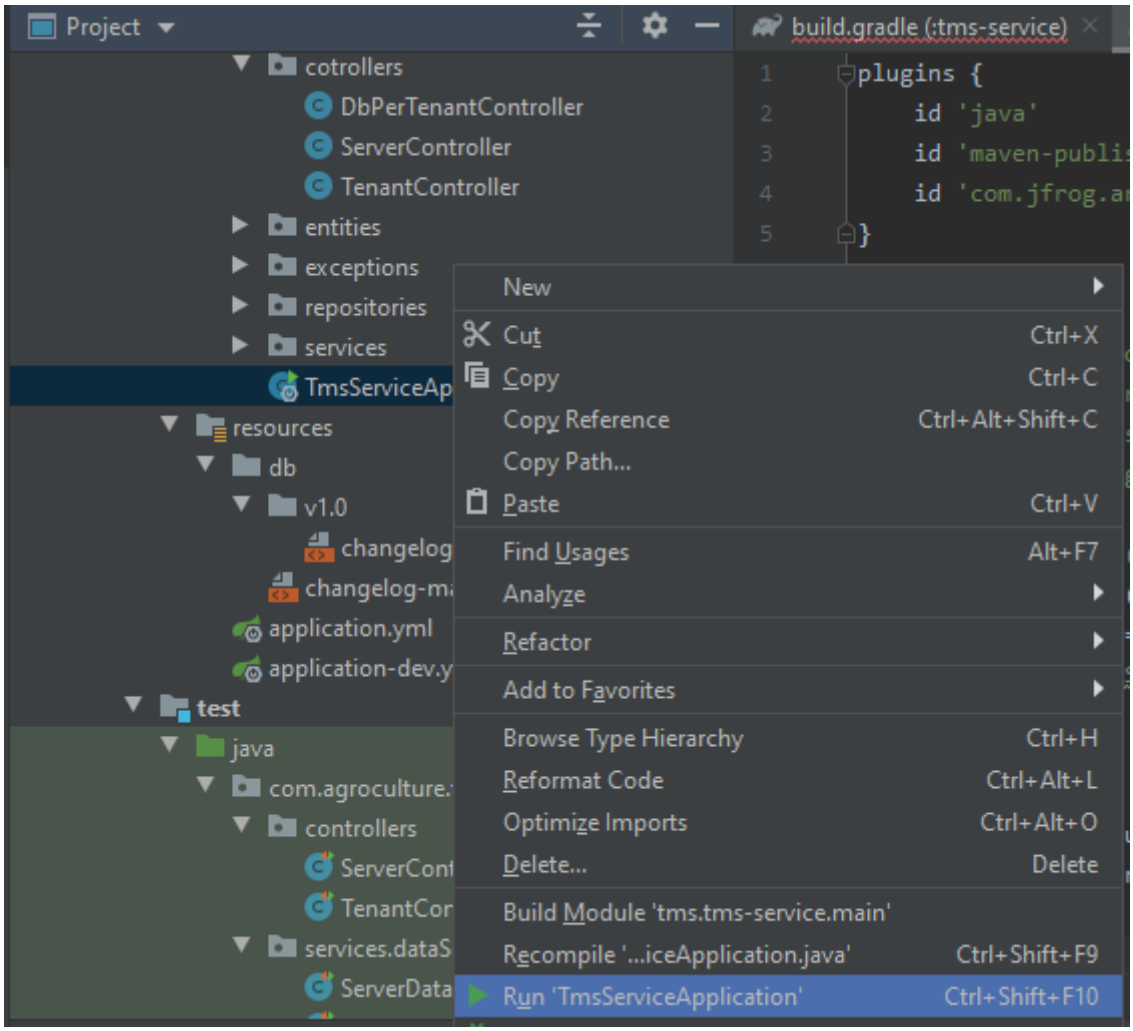

Рисунок 5.2.5 — Запуск Spring Boot проекту

На цьому етапі буде піднятий сервіс tms та створена структура бази даних. Після цього необхідіно з’єднатись з базою tms та виконати наступні команди (таблиця 5.2.1).

Таблиця 5.2.1 – Заповнення бази даних TMS

INSERT INTO servers VALUES ('1', 'jdbc:postgresql://localhost:5432', 'postgres', 'password', 1, true); **INSERT INTO servers VALUES ('2', 'jdbc:postgresql://localhost:5433', 'postgres', 'password', 1, true);** **INSERT INTO tenant VALUES (1, 'TEF', null, 'HEAT POWER ENGINEER**

```
DEPARTMENT', true);
    INSERT INTO tenant VALUES (2, 'WELD', null, 'WELDING DEPARTMENT', true);

    INSERT INTO db_pertenant VALUES (1, 'TEF', 'TEF', 'user-service', '1', 'postgres', 'org.postgresql.Driver', 'password', true);
    INSERT INTO db_pertenant VALUES (2, 'WELD', 'WELD', 'user-service', '2', 'postgres', 'org.postgresql.Driver', 'password', true);
```

Тепер tenant-management-service працює та повертає дані.

5.3. Налаштування user-service-facade

Далі необхідно перейти до папки **D:/services** та виконати команду **git clone https://github.com/Jayter/user-service-facade.git** (рисунок 5.3.1).

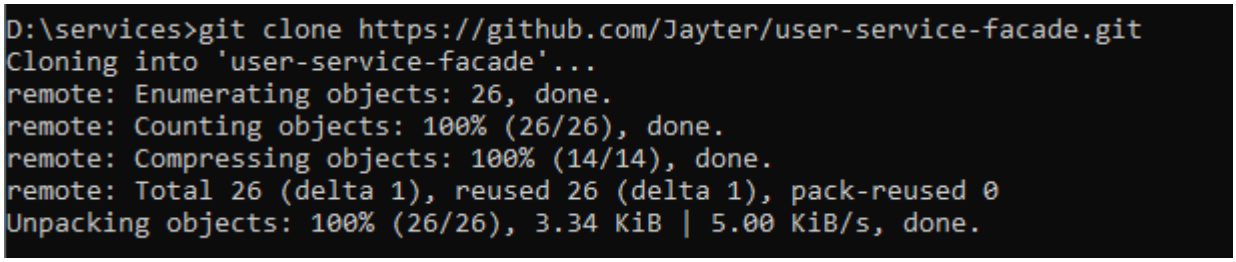

Рисунок 5.3.1 — Скачування проекту user-service-facade

Далі переходимо до папки **user-service-facade** та виконати команду **mvn install spring-boot:run** (рисунок 5.3.2). Ця команда інсталює проект та запустить сервіс user-service-facade.

Рисунок 5.3.2 — Інсталяція та запуск сервісу user-service-facade

Після успішного запуску ми побачимо в логах повідомлення про те, що сервіс успішно запущено (рисунок 5.3.3).

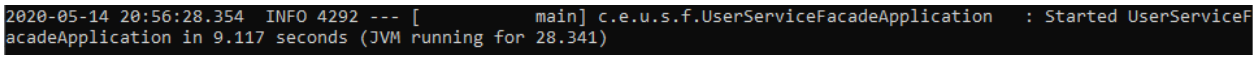

Рисунок 5.3.2 — Повідомлення про успішний запуск сервісу

5.4. Налаштування user-service-client

Далі необхідно перейти до папки **D:/services** та виконати команду **git clone https://github.com/Jayter/user-service.git** (рисунок 5.4.1).

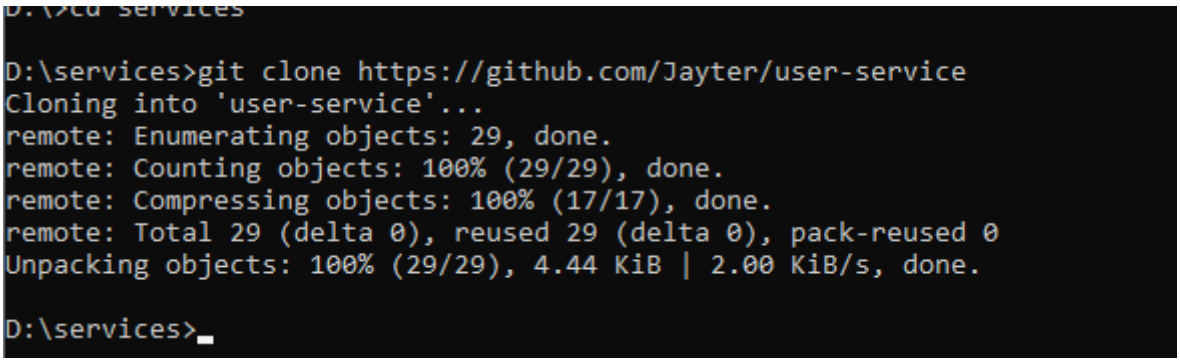

Рисунок 5.4.1 — Скачування проекту user-service

Далі переходимо до папки **user-service** та виконати команду **mvn install spring-boot:run -Drun.profiles=TEF** (рисунок 5.4.2). Ця команда інсталює проект та запустить сервіс user-service з клієнтом TEF.

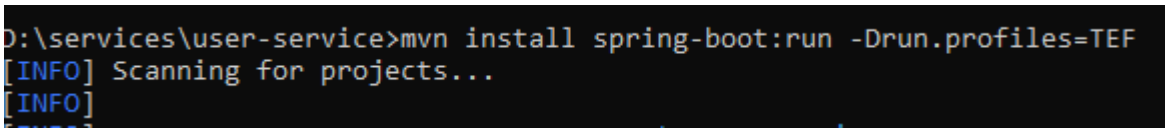

Рисунок 5.4.2 — Інсталяція та запуск проекту user-service для TEF

Далі відкриваємо браузер та переходимо на сторінку http://localhost:8083. Ми побачимо головну сторінку для сервісу user-service-TEF (рисунок 5.4.3).

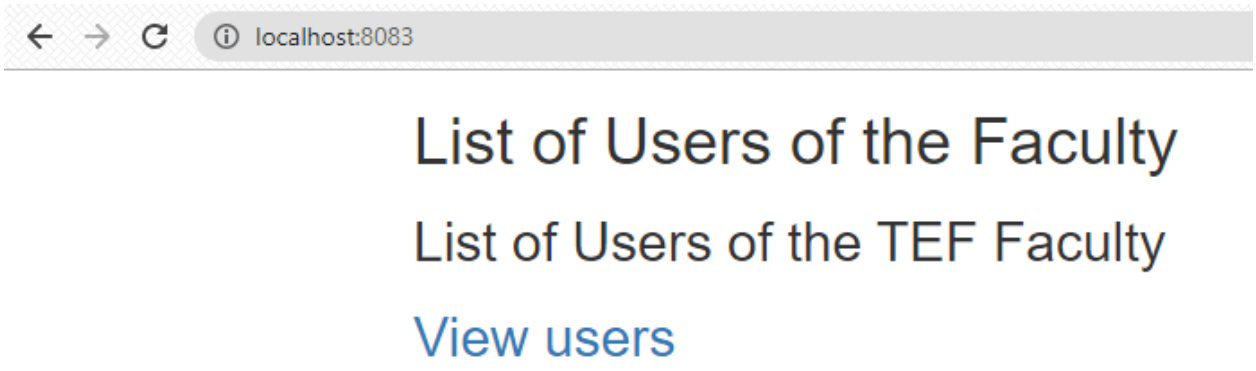

Рисунок 5.4.3 — Головна сторінка user-service-TEF

Аналогічним чином відкриваємо іншу командний рядок, переходимо в папку D:/services/user-service та виконуємо команду **mvn spring-boot:run -Drun.profiles=WELD**. Ця команда запустить сервіс user-service з клієнтом WELD. Далі відкриваємо браузер та переходимо на сторінку http://localhost:8084. Ми побачимо головну сторінку для сервісу user-service-WELD (рисунок 5.4.4).

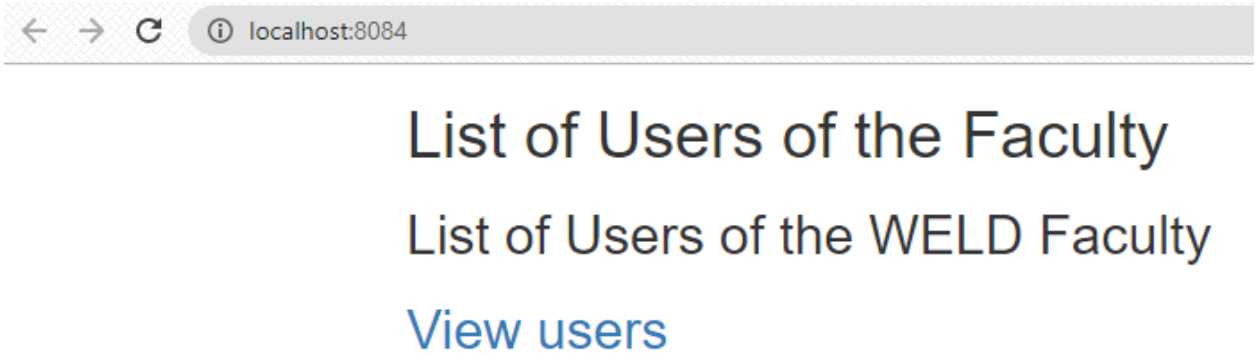

Рисунок 5.4.4 — Головна сторінка user-service-TEF

5.5. Демонстрація роботи бібліотеки

На головній сторінці TEF натиснемо **View users**. Ми отримаємо список студентів та дані про них (рисунок 5.5.1).

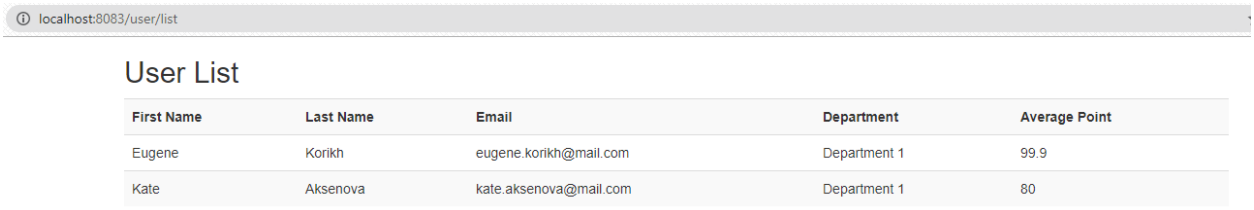

Рисунок 5.5.1 — Головна сторінка user-service-TEF

Тепер порівняємо дані з базою **user-service:5432** (рисунок 5.5.2).

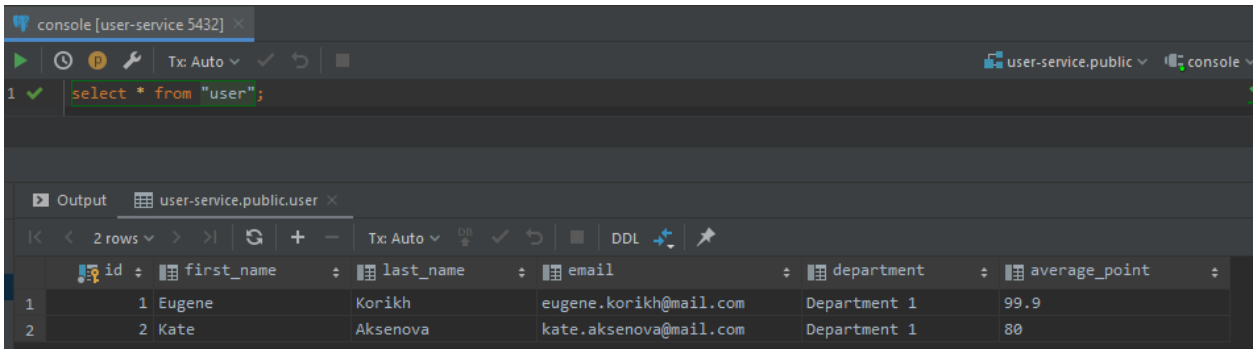

Рисунок 5.5.2 — Дані з бази user-service:5432

Тепер ми переконані, що отримуємо дані з правильної бази. Аналогічним чином виконуємо первірку для WELD (рисунок 5.5.3 та 5.5.4).

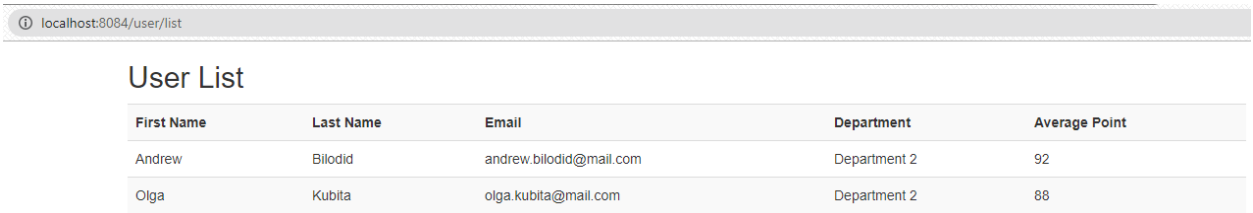

Рисунок 5.5.3 — Головна сторінка user-service-WELD

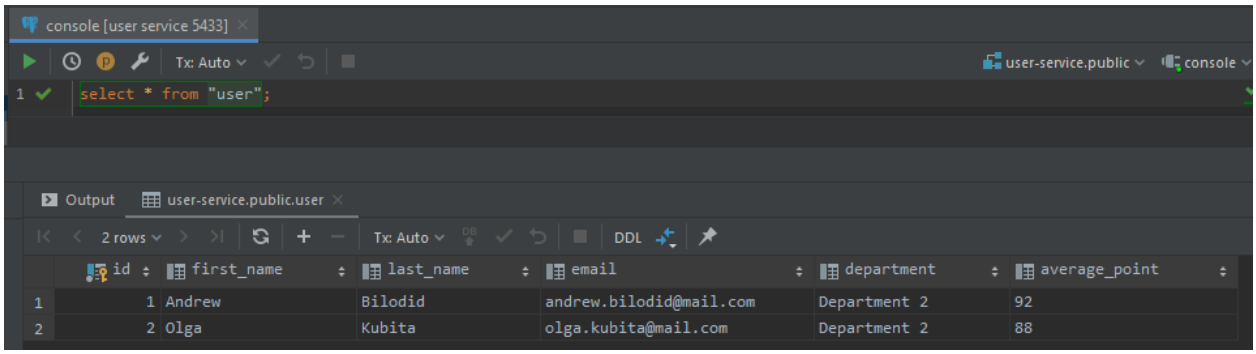

Рисунок 5.5.4 — Дані з бази user-service:5433

Тепер ми переконані, що дані співпадають. Це відбувається завдяки тому, що tenant-connection-resolver використовує роутінг між факультетом та базою, отриманий від сервісу tms (рисунок 5.5.5).

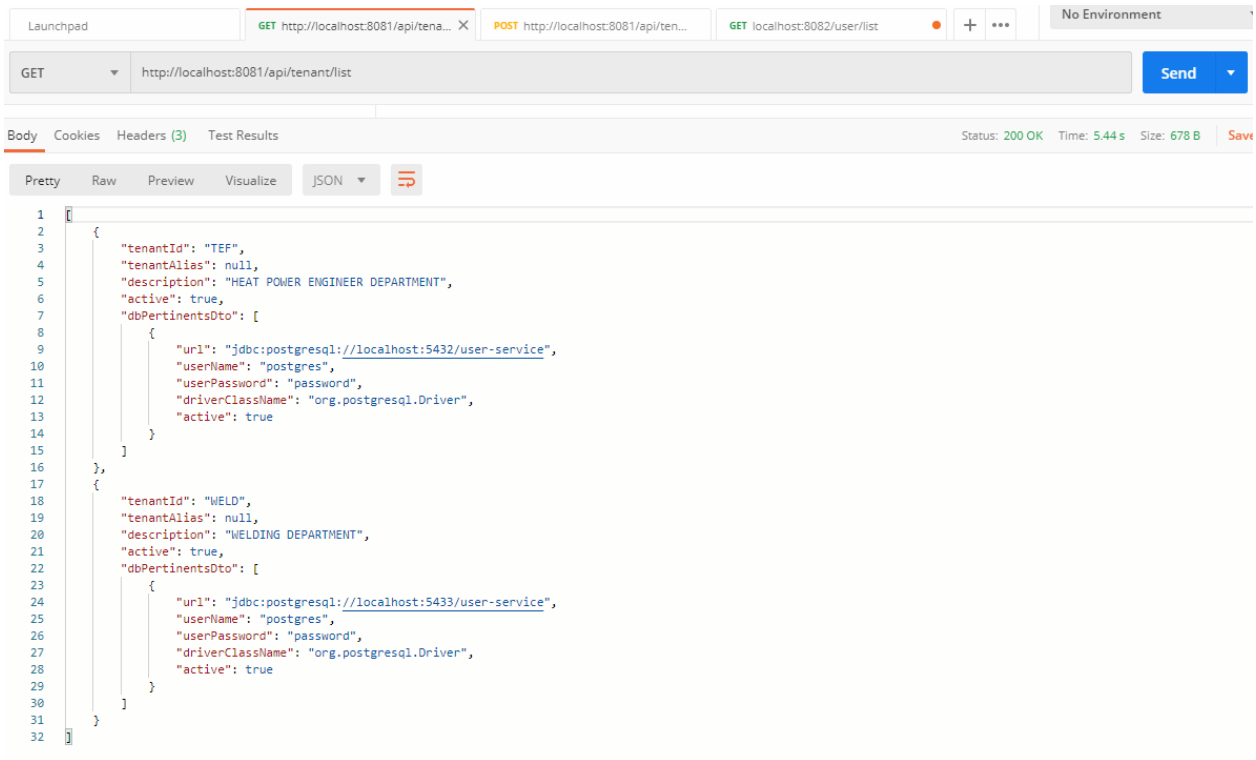

Рисунок 5.5.5 — Роутінг між факультетом та базою від tms

ВИСНОВКИ

У даній дипломній роботі було проаналізовано види мультиклієнтності, їх переваги та недоліки. Було пояснено різницю між мультиклієнтністю та мультиекземплярністю.

На основі проведеного аналізу було створено бібліотеку, що аналізує поточного клієнта та забезпечує з’єднання з необхідною базою даних. Також була створена архітектура та розроблена система для демонстрації роботи бібліотеки. Це програмне забезпечення являє собою серверний додаток, створений на мові Java, використовуючи Java 8 та Spring Boot. Також для зручної роботи з системою було розроблено відповідний веб інтерфейс. Ця система допоможе створювати зручні однотипні програми для різних факультетів, залишаючи базу даних на управління факультету.

Таким чином, дана бібліотека дає можливість створювати програмні продукти для усіх факультетів університету одночасно. При цьому один працюючий екземпляр продукту може обслуговувати кожного з них незалежно. При цьому кожен з факультетів матиме свою особисту базу даних.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Krebs, Rouven «Architectural Concerns in Multi-tenant SaaS Applications» [Електронний ресурс] — Krebs, Rouven, 2012 — Режим доступу: http://se2.informatik.uni-wuerzburg.de/pa/uploads/papers/paper-371.pdf .

2. «Defining the true meaning of cloud» [Електронний ресурс] / Wainewright, Phil, 2010 — Режим доступу: https://www.zdnet.com/article/defining-the-true-meaning-of-cloud/?tag=mantle_skin;content.

3. «Cloud Architecture Patterns: Using Microsoft amit [Електронний ресурс] / Wilder, Bill, 2012. Режим доступу: https://books.google.com/books?id=X-d6JVHQwo8C.

4. «Extensibility and data sharing in evolving multi-tenant databases» [Електронний ресурс] / Aulbach, S, 2011 — Режим доступу: https://doi.org/10.1109%2FICDE.2011.5767872.

5. «Multi-Tenant Fair Share in NoSQL Data Stores [Електронний ресурс] / Zeng, Jiaan, 2014 — Режим доступу: https://doi.org/10.1109%2FCLUSTER.2014.6968761.

6. «Spring 5 для профессионалов» [Електронний ресурс] / Ю. Козмина, Р. Харроп, К. Шефер, К. Хо, 2019 Режим доступу: http://www.williamspublishing.com/Books/978-5-907114-07-4.html.

7. Раджпут Д. «Spring. Все паттерны проектирования» / Раджпут Д. — К. : «Питер», 2019. — 320 с.

8. Лонг Д., Бастани К. «Java в облаке. Spring Boot, Spring Cloud, Cloud Foundry» / Лонг Д., Бастани К. — К. : «Питер», 2019. — 624 с.

9. Кей С. Хорстманн. «Java. Библиотека профессионала, том 1. Основы. 10-е издание» / Кей С. Хорстманн — М. : «Вильямс», 2017. — 864 с.

10. Джошуа Блох. «Java. Эффективное программирование» / Джошуа Блох — К. : «Диалектика», 2019. — 464 с.

Додаток А

Код бібліотеки tenant-connection-resolver

```
@RequiredArgsConstructor
public class JpaConfiguration {

   private final TenantInitializationService initializationService;

   private final DbPerTenantDtoToDataSourceConverter converter;

   @Bean
   public DataSource dataSource() {
      AbstractRoutingDataSource dataSource = new
TenantAwareRoutingService();

      Map<Object, Object> targetDataSources
            = initializationService.getDataSourcesByTenant()
                        .entrySet()
                        .stream()
                        .collect(toMap(Entry::getKey, e ->
converter.convert(e.getValue())));

      dataSource.setTargetDataSources(targetDataSources);

      dataSource.afterPropertiesSet();
```

```
        return dataSource;
    }
}

@Configuration
@EnableFeignClients(clients = TenantEndpoint.class)
@Import(value = {JpaConfiguration.class,
TenantConnectionResolverWebMvcConfigurer.class})
public class TenantConnectionResolverConfiguration {

    @Bean
    public DbPerTenantDtoToDataSourceConverter converter() {
        return new DbPerTenantDtoToDataSourceConverter();
    }

    @Bean
    public TenantInitializationService initializationService(TenantEndpoint
endpoint) {
        return new TenantInitializationService(endpoint);
    }

    @Bean
    public TenantRequestInterceptor requestInterceptor() {
        return new TenantRequestInterceptor();
    }
}
```

```
@Configuration
@RequiredArgsConstructor
public class TenantConnectionResolverWebMvcConfigurer implements
WebMvcConfigurer {

   private final TenantRequestInterceptor requestInterceptor;

   @Override
   public void addInterceptors(InterceptorRegistry registry) {
      registry.addInterceptor(requestInterceptor);
   }

   @Override
   public void addResourceHandlers(ResourceHandlerRegistry
resourceHandlerRegistry) {

   }

   @Override
   public void addCorsMappings(CorsRegistry corsRegistry) {

   }

   @Override
   public void addViewControllers(ViewControllerRegistry
```

```
viewControllerRegistry) {

    }

    @Override
    public void configureViewResolvers(ViewResolverRegistry
viewResolverRegistry) {

    }

    @Override
    public void
addArgumentResolvers(List<HandlerMethodArgumentResolver> list) {

    }

    @Override
    public void
addReturnValueHandlers(List<HandlerMethodReturnValueHandler>
list) {

    }

    @Override
    public void
configureMessageConverters(List<HttpMessageConverter<?>> list) {
```

```
    }

    @Override
    public void
extendMessageConverters(List<HttpMessageConverter<?>> list) {

    }

    @Override
    public void
configureHandlerExceptionResolvers(List<HandlerExceptionResolver>
list) {

    }

    @Override
    public void
extendHandlerExceptionResolvers(List<HandlerExceptionResolver> list)
{

    }

    @Override
    public Validator getValidator() {
        return null;
```

```
    }

    @Override
    public MessageCodesResolver getMessageCodesResolver() {
        return null;
    }

    @Override
    public void
configureContentNegotiation(ContentNegotiationConfigurer configurer) {
        configurer.favorPathExtension(false);
    }

    @Override
    public void configureAsyncSupport(AsyncSupportConfigurer
asyncSupportConfigurer) {

    }

    @Override
    public void
configureDefaultServletHandling(DefaultServletHandlerConfigurer
defaultServletHandlerConfigurer) {

    }
```

```
    @Override
    public void addFormatters(FormatterRegistry formatterRegistry) {

    }

    @Override
    public void configurePathMatch(PathMatchConfigurer configurer) {
        configurer.setUseSuffixPatternMatch(false);
    }
}

public class TenantContext {

    private static final ThreadLocal<String> CONTEXT = new
ThreadLocal<>();

    public static void setTenantId(String tenantId) {
        CONTEXT.set(tenantId);
    }

    public static String getTenantId() {
        return CONTEXT.get();
    }

    public static void clear() {
        CONTEXT.remove();
```

```
    }
}

public class DbPerTenantDtoToDataSourceConverter {

    public DataSource convert(DbPerTenantDto dto) {
        return DataSourceBuilder.create()
                        .driverClassName(dto.getDriverClassName())
                        .username(dto.getUserName())
                        .password(dto.getUserPassword())
                        .url(dto.getUrl())
                        .build();
    }
}

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class DataSourceConfig {

    private String url;
    private String username;
    private String password;
    private String driverClassName;
```

```
}

public class UnknownTenantException extends RuntimeException {

   public UnknownTenantException(String message) {
      super(message);
   }
}

public class TenantAwareRoutingService extends
AbstractRoutingDataSource {

   @Override
   protected Object determineCurrentLookupKey() {
      return TenantContext.getTenantId();
   }
}

@FeignClient(name = "TenantEndpoint", url = "${tenant-connection-
resolver.url}")
public interface TenantEndpoint {

   @RequestMapping(method = RequestMethod.GET)
   List<TenantDto> getAllTenants();
}
```

```
@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class DbPerTenantDto {

    private String url;
    private String userName;
    private String userPassword;
    private String driverClassName;
    private boolean active;

}

@Getter
@Setter
@AllArgsConstructor
@NoArgsConstructor
public class TenantDto {

    private String tenantId;
    private String tenantAlias;
    private String description;
    private boolean active;

    private List<DbPerTenantDto> dbPertinentsDto;
```

```
}

@RequiredArgsConstructor
public class TenantInitializationService {

   private final TenantEndpoint endpoint;

   @Value("${tenant-connection-resolver.db-name}")
   private String dbName;

   public Map<String, DbPerTenantDto> getDataSourcesByTenant() {
      return endpoint.getAllTenants()
                  .stream()
                  .collect(HashMap::new,
                        (m, dto) -> m.put(dto.getTenantId(),
dto.getDbPertinentsDto()
                                                .stream()
                                                .filter(db ->
db.getUrl().endsWith(dbName))
                                                .findFirst()
                                                .orElse(null)),
HashMap::putAll);
   }
}

public class TenantRequestInterceptor extends
```

```
HandlerInterceptorAdapter {

    @Value("${tenant-connection-resolver.tenant-header}")
    private String tenantHeader;

    @Override
    public boolean preHandle(HttpServletRequest request,
HttpServletResponse response, Object handler) {
        String tenantId = request.getHeader(tenantHeader);
        TenantContext.setTenantId(tenantId);
        return true;
    }

    @Override
    public void afterCompletion(HttpServletRequest request,
HttpServletResponse response, Object handler, Exception ex) {
        TenantContext.clear();
    }
}
```

Додаток Б

Код сервісу user-service-facade

```
@Configuration
@Import(TenantConnectionResolverConfiguration.class)
public class UserServiceFacadeConfiguration {


}


@Controller
@RequestMapping("/user")
@RequiredArgsConstructor
public class UserController {

    private final UserService userService;

    @ResponseBody
    @GetMapping(value = "/list")
    public List<UserEntity> list() {
        return userService.listAll();
    }
}


@Getter
@Setter
@Entity
```

```
@Table(name = "user", schema = "public")
public class UserEntity {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "id")
    private Long id;

    @Column(name = "first_name")
    private String firstName;

    @Column(name = "last_name")
    private String lastName;

    @Column(name = "email")
    private String email;

    @Column(name = "department")
    private String department;

    @Column(name = "average_point")
    private BigDecimal averagePoint;
}

@Repository
public interface UserEntityRepository extends JpaRepository<UserEntity,
```

```
Long> {

}

@Service
@RequiredArgsConstructor
public class UserService {

    private final UserEntityRepository repository;

    public List<UserEntity> listAll() {
        return repository.findAll();
    }

}

@SpringBootApplication
public class UserServiceFacadeApplication {

    public static void main(String[] args) {
        SpringApplication.run(UserServiceFacadeApplication.class, args);
    }
}

tenant-connection-resolver:
  tenant-header: tenant
```

```
  db-name: user-service
  url: localhost:8081/api/tenant/list

server:
  port: 8082

spring:
  jpa:
    open-in-view: false
    show-sql: true
    hibernate.naming.physical-strategy:
org.hibernate.boot.model.naming.PhysicalNamingStrategyStandardImpl
    database: postgresql
    hibernate:
      ddl-auto: none
  datasource:
    initialize: false
```

Додаток В

Код сервісу user-service-client

```
@FeignClient(name = "user-endpoint", url = "${user-service.facade-
url}/user")
public interface UserEndpoint {

   @Headers("tenant: {}")
   @RequestMapping(value = "/list", method = RequestMethod.GET)
   List<User> getAllUsers(@RequestHeader("tenant") String language);
}

@Controller
public class IndexController {

   @Value("${user-service.tenant}")
   private String tenant;

   @RequestMapping("/")
   public String index(Model model) {
      model.addAttribute("tenant", tenant);
      return "index";
   }
}

@Controller
```

```
@RequestMapping("/user")
@RequiredArgsConstructor
public class UserController {

    private final UserService userService;

    @GetMapping(value = "/list")
    public String getAllUsers(Model model) {
        model.addAttribute("users", userService.getAllUsers());
        return "users";
    }
}

@Getter
@Setter
public class User {

    private Long id;

    private String firstName;

    private String lastName;

    private String email;

    private String department;
```

```
    private BigDecimal averagePoint;
}

@Service
@RequiredArgsConstructor
public class UserService {

    @Value("${user-service.tenant}")
    private String tenant;

    private final UserEndpoint userEndpoint;

    public List<User> getAllUsers() {
        try {
            return userEndpoint.getAllUsers(tenant);
        } catch (Exception e) {
            return Collections.emptyList();
        }
    }
}

@EnableFeignClients
@SpringBootApplication
public class UserServiceApplication {
```

```
    public static void main(String[] args) {
        SpringApplication.run(UserServiceApplication.class, args);
    }
}

user-service:
  facade-url: localhost:8082
  tenant: TEF

server:
  port: 8083

user-service:
  facade-url: localhost:8082
  tenant: WELD

server:
  port: 8084
```